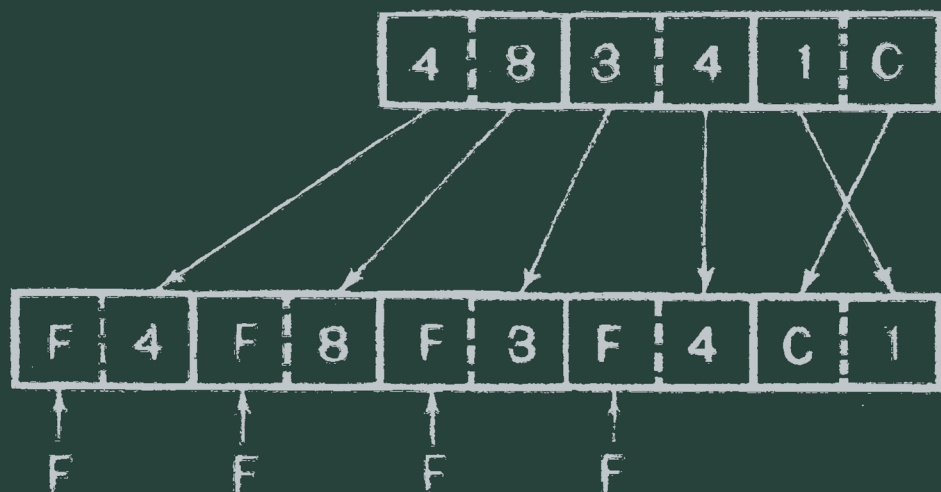


YU. MITNIK, A. KHMELNITSKY

# Programming and Algorithmic Languages



MIR PUBLISHERS MOSCOW







# **PROGRAMMING AND ALGORITHMIC LANGUAGES**

**Ю. Ш. Митник, А. С. Хмельницкий**

**ПРОГРАММИРОВАНИЕ  
И АЛГОРИТМИЧЕСКИЕ ЯЗЫКИ**

**Москва «Машиностроение»**

YU. MITNIK, A. KHMELNITSKY

# Programming and Algorithmic Languages



MIR PUBLISHERS MOSCOW

First published 1987  
Revised from the 1984 Russian edition  
Translated from the Russian by Peter I. Zabolotny

---

#### TO THE READER

Mir Publishers welcome your comments on the contents, translation, and design of the book.

We would also be pleased to receive any suggestions you care to make about our future publications.

Our address is:  
USSR, 129820,  
Moscow, I-110, GSP,  
Pervy Rizhsky Pereulok, 2,  
Mir Publishers

---

*На английском языке*

*Printed in the Union of Soviet Socialist Republics*

© Издательство «Машиностроение», 1984

© English translation, Mir Publishers, 1987

# CONTENTS

<b>Preface</b> . . . . .	8
<b>PART ONE. FUNDAMENTALS OF PROGRAMMING</b> . . . . .	9
<b>Chapter 1. Computer Arithmetic and Logic Concepts</b> . . . . .	9
1.1. Number Systems . . . . .	9
1.2. Binary Coding of Decimal Numerals . . . . .	13
1.3. Representation of Numbers in Computer . . . . .	15
1.4. Representing Negative Numbers in Digital Computers. True Form, Ones-Complement Form, and Twos-Complement Form. Modified Codes . . . . .	17
<b>Chapter 2. Computer Organization</b> . . . . .	20
2.1. General Concept of the Third-Generation Computer . . . . .	20
2.2. Data Representation in the Main Storage . . . . .	25
2.3. Classification of Storage Devices . . . . .	32
2.4. Characteristics of Storage Devices . . . . .	34
2.5. Magnetic Disk Drive. General . . . . .	35
2.6. Applications and Technical Data of Magnetic Tape Units . . . . .	36
2.7. Data Organization in the Magnetic Tape Unit . . . . .	38
2.8. Fundamentals of the ES EVM Input/Output Devices . . . . .	39
2.9. Operator's Console . . . . .	42
2.10. Concept of Input/Output Interface . . . . .	42
2.11. Interface. General . . . . .	45
<b>Chapter 3. Programming, General</b> . . . . .	45
3.1. Algorithm Concept . . . . .	45
3.2. Specification of Algorithm . . . . .	48
3.3. Computation Process According to Block Diagram of Digital Computer . . . . .	51
<b>PART TWO. ALGORITHMIC LANGUAGES</b> . . . . .	53
<b>Chapter 4. Instruction Set</b> . . . . .	53
4.1. Instruction Format . . . . .	53
4.2. Arithmetic Fixed-Point Operations . . . . .	55
4.3. Programming Techniques . . . . .	59

4.4.	Branching . . . . .	63
4.5.	Programming Branching Algorithms . . . . .	67
4.6.	Subroutines, Linkage Conventions . . . . .	73
4.7.	Programming Loop Algorithms . . . . .	84
4.8.	Short and Long Numbers . . . . .	97
4.9.	Floating-Point Arithmetic Operations . . . . .	102
4.10.	Operations on Data Codes . . . . .	111
4.11.	Decimal Arithmetic Instructions . . . . .	126
4.12.	Conversion of Number Format . . . . .	141
4.13.	Edit Instructions . . . . .	147
4.14.	Translation Instructions . . . . .	155
4.15.	The Execute Instruction . . . . .	157
<b>Chapter 5.</b>	<b>Control . . . . .</b>	<b>159</b>
5.1.	Control Programs . . . . .	159
5.2.	Supervisor Work Supports . . . . .	161
5.3.	Program Status Word (PSW) . . . . .	163
5.4.	Storage Protection . . . . .	165
5.5.	Program Interrupts . . . . .	167
5.6.	Supervisor Call . . . . .	170
<b>Chapter 6.</b>	<b>Channel Organization and Input/Output . . . . .</b>	<b>171</b>
6.1.	Information Interchange Principles . . . . .	171
6.2.	Channel Programs . . . . .	174
6.3.	Channel-CPU Interaction . . . . .	179
6.4.	Status Bytes . . . . .	184
6.5.	Input/Output Interrupts . . . . .	187
6.6.	Input/Output Instructions . . . . .	189
6.7.	Initial Program Loading (IPL) . . . . .	191
<b>Chapter 7.</b>	<b>Introduction to Assembler . . . . .</b>	<b>192</b>
7.1.	Symbolic Programming . . . . .	192
7.2.	Writing Symbolic Statements . . . . .	194
7.3.	Assembler Alphabet, Terms and Expressions . . . . .	196
7.4.	Machine Instructions . . . . .	202
7.5.	Extended Mnemonics . . . . .	206
7.6.	USING and DROP Statements . . . . .	208
7.7.	Defining Constants . . . . .	210
7.8.	Address Constants and Defining Storage . . . . .	215
7.9.	Program Sectioning . . . . .	221
7.10.	Macros . . . . .	224
7.11.	Sample Programs . . . . .	232
<b>Chapter 8.</b>	<b>Introduction to PL/I . . . . .</b>	<b>236</b>
8.1.	Problem-Oriented Programming Languages . . . . .	236
8.2.	PL/I Character Set . . . . .	241
8.3.	Writing a PL/I Program . . . . .	242
8.4.	PL/I Syntax . . . . .	243
8.5.	Data . . . . .	246
8.6.	Operations on Data . . . . .	254
8.7.	Expressions and Assignment Statements . . . . .	260
8.8.	Arrays and Operations on Them . . . . .	261

8.9. GOTO Statement. IF Statement . . . . .	268
8.10. DO-Loop . . . . .	273
8.11. Edit-Directed I/O . . . . .	283
<b>Chapter 9. Introduction to FORTRAN . . . . .</b>	<b>300</b>
9.1. Special Features of FORTRAN . . . . .	300
9.2. Basic Elements of the Language . . . . .	301
9.3. Numeric Constants . . . . .	303
9.4. Simple Variables . . . . .	304
9.5. Arithmetic Operations and Expressions . . . . .	304
9.6. Elementary Mathematical Functions . . . . .	306
9.7. Arithmetic Assignment . . . . .	307
9.8. Writing a Program on Coding Sheet and Punching It into Cards . . . . .	307
9.9. Input of Numeric Data . . . . .	308
9.10. Information Printout . . . . .	310
9.11. Sample Examples of Simple Programs in FORTRAN . . . .	311
9.12. GOTO Statement . . . . .	312
9.13. Arithmetic IF Statement . . . . .	312
9.14. Variable Arrays . . . . .	313
9.15. DO Statement . . . . .	315
9.16. FUNCTION Subprogram . . . . .	316
<b>Chapter 10. Introduction to Operating System DOS/ES . . . . .</b>	<b>319</b>
10.1. DOS Components and Structure . . . . .	319
10.2. Basic Concepts of DOS . . . . .	320
10.3. Job Control Program . . . . .	322
10.4. Linkage-Editor . . . . .	324
<b>References . . . . .</b>	<b>326</b>
<b>Index . . . . .</b>	<b>327</b>

## PREFACE

In a short quarter-century, the electronic computer has had an enormous impact upon business, industry, science, education, and society in general. Scarcely an occupation or academic discipline has not been profoundly affected by the computer's speed and tireless capacity for work. The advent of large-scale integrated circuits, microprocessors, and large-capacity semiconductor memories have made possible powerful computer systems, cutting their cost, dimensions, and power consumption.

In order for automation to be introduced throughout the economy, a common engineering base is essential. The CMEA countries, therefore, have started mass production of a Unified System of Computers, the Cyrillic ES EVM title yielding the abbreviation which we shall now use.

The six base models differ in their specifications (speed, main storage capacity, etc.) and some 140 different peripherals are available. These peripherals substantially enhance the abilities of the ES EVM computers without affecting their software compatibility.

The ES EVM computer system has become a vital tool for any research worker who must solve complicated problems. They have been applied in the most diverse fields of science and technology, such as physics, chemistry, geology, biology, nuclear power engineering, space research, rocket technology, aviation, machine-tool industry, and management. There is hardly any industry which does not use or cannot use computers in some way.

The software for the ES range includes operating systems, automatic programming aids in standard languages and compilers.

Algorithmic languages are the most important part of the ES EVM software, and provide tools for solving problems on ES computers.

All those attending the computers, system programmers and operators, those involved in data processing, and computer maintenance specialists must be conversant with programming in such problem-oriented languages as ALGOL, FORTRAN, COBOL, Assembler, and PL/I.

This book is aimed at training such specialists of medium level. We believe this text will be of help to those using an independent approach to the study of computer programming and algorithmic languages.

## PART 1

# Fundamentals of Programming

## CHAPTER 1

### COMPUTER ARITHMETIC AND LOGIC CONCEPTS

#### 1.1. Number Systems

The *number (numeration) system* is a set of techniques and rules for representing numbers with the help of which a unique relationship can be established between any number and its representation in the form of a set of a finite number of symbols. The set of symbols used for this representation is known as *digits*.

The history of counting knows many diverse systems of numeration. However, an analysis of various number systems allows all the systems to be classified into two groups, positional and nonpositional, according to whether a digit value depends upon its location with respect to other digits.

*Nonpositional* are number systems in which the value of a given symbol never changes, regardless of its place or position in a given number. The known nonpositional systems of numeration employ either the addition principle (such systems are called additive systems), or the multiplication principle (such systems are known as multiplicative ones). The most simple is a number system consisting of one digit, a 1. An example is a sequence of strokes *////////*, in which the quantitative equivalent (weight) of each stroke (character) corresponds to one, and which represents a numeral for eight. Systems like this one are not used nowadays.

The Roman number system is a more intricate not 'place value' system which utilizes principles both of addition and subtraction. If a numeral of less quantitative equivalent appears after a numeral having a greater quantitative equivalent, their quantitative equivalents (weights) are added. If it appears before a greater numeral, their weights are subtracted. For example, in the Roman numeral IV, I means 1 and V means 5. When I appears before V, the number is 4 ( $5 - 1$ ). For the numeral VI, I still has a value of 1, and V a value of 5; but the number has a value of 6 ( $5 + 1$ ) because of the position of the numeral I. However, V always means 5. Nowadays, the Roman system of numeration is not used in the computation practice. It finds applications in various designations, such as numbering book chapters, marking hours in the face of clocks and watches,

etc. The not 'place value' (nonpositional) systems feature very intricate and cumbersome algorithms of number representation and execution of arithmetical operations of addition, subtraction, multiplication, and division, for which reason they are not used in the digital computers.

Used in the digital computers are *positional* ('place value') number systems. These are positional representation systems in which digits have weights which are multiplier values that depend upon their digit position in relation to the other digits of a number. In such systems as these, the weight (value) of each digit depends upon its position in the number representation. For example, in the number 575 the digit 5 is encountered twice. Its weight, however, is not the same in both cases: the least significant digit 5 occupies the units position and has a value of 5. The most significant digit 5 is in the hundreds position and stands for 500.

The most ancient of the known positional systems is the sexadecimal Babylonian system of numeration in which there were only two symbols used to represent one and ten. These two symbols were used repeatedly to write numerals from 1 to 59. Traces of this system remain till now. Partially it is used in time counting (an hour contains 60 min, a minute comprises 60 seconds), and in degree measurement of angles ( $1^\circ = 60'$ ,  $1' = 60''$ ). In addition to the Babylonian system, we know a base-20 system of numeration which was used by American Indian peoples, and is best known in the well-developed Mayan number system. Celtic traces of a base 20 were found in the French monetary system in which the basic unit of money, a franc, was a 20-sou unit.

Our present number system that has found most wide use, is also positional. Compared with the nonpositional systems, the major advantage of the positional number systems is in ease of number representation and simplicity of arithmetic operations.

An analysis of the decimal number system shows us that this system employs the principles of multiplication and addition. For example, consider the number 456. The '4' on the left means 4 hundreds and may be written as  $4 \times 10 \times 10$  (since  $100 = 10 \times 10$ ). The '5', the second digit from the left, means 5 tens and may be written as  $5 \times 10$ , and the '6', the third digit from the left, means 6 unities, and may be written as  $6 \times 1$ . Thus 456 can be represented in the form of the following expanded numeral (polynomial):  $456 = 4 \cdot 100 + 5 \cdot 10 + 6 \cdot 1$ , or  $456 = 4 \cdot 10^2 + 5 \cdot 10^1 + 6 \cdot 10^0$ .

All the positional systems of numeration feature a certain number known as the *base* of the number system. This is the amount by which a digit is multiplied or divided when moved to an adjacent digit position. In the decimal system, for example, a '3' becomes '30' if moved one digit position to the left, and '0.3' if moved a digit position to the right.

In any positional number system with a base  $p$  ( $|p| \geq 1$ ), the base of this system does not change whether we use ten or four or any number as a base. For example, the numeral '10' usually means 1 ten, when the base is 10, but if the base were eight, it would mean 1 eight, and if the base were two, it would mean 1 two, and so on. Remember that the symbol '10' always represents the base, since '1' in the next-to-last place indicates the base taken once. Further, we shall consider solely positional number systems employing the principles of multiplication and addition. In such systems with base  $p$ , an arbitrary number  $N$  will take the form

$$N = a_{n-1}p^{n-1} + \dots + a_1p^1 + a_0p^0 + a_{-1}p^{-1} + \dots + a_{-m}p^{-m} \quad (1.1)$$

or

$$N = \sum_{i=-m}^{n-1} a_i p^i$$

where  $p$  is a number system base representing an integer greater than 1 in absolute value,  $a_i$  are digits from the set  $0, 1, 2, \dots, p-1$ ,  $m$  is the number of digit places to the right of the point (the fractional part of the number), and  $n$  is the number of digit places to the left of the point (the integral part of the number).

The generally used decimal system of numeration fully satisfies the relation (1.1), i.e. an arbitrary number  $N$  in the decimal number system can be represented as follows

$$N = a_{n-1}10^{n-1} + a_{n-2}10^{n-2} + \dots + a_010^0 + a_{-1}10^{-1} + \dots + a_{-m}10^{-m} \quad (1.2)$$

or

$$N = \sum_{i=-m}^{n-1} a_i 10^i$$

Therefore, the generally used number representation is nothing more, than an enumeration of the coefficients of an expanded numeral (polynomial) shown in (1.1).

In writing a number, the number system is designated by putting the number base as a subscript which represents the base of the corresponding number system, the base being written in the decimal system. For example,  $763_8$  means that the number is in the octal numbering system.

**Example.** Write expanded numerals for the following numbers  $789.35_{10}$  and  $1101.0101_2$ .  $N = 789.35_{10} = 7 \cdot 10^2 + 8 \cdot 10^1 + 9 \cdot 10^0 + 3 \cdot 10^{-1} + 5 \cdot 10^{-2}$ ;  $N = 1101.0101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4}$ .

In the construction of computers and development of programming methods, use is made of the positional (place value) number systems with base 2, 8 and 16.

In the binary number system, an arbitrary number  $N$  will be as follows:

$$N = a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \dots + a_02^0 + a_{-1}2^{-1} + \dots + a_{-m}2^{-m} \quad (1.3)$$

i.e.

$$N = \sum_{i=-m}^{n-1} a_i 2^i$$

where  $a_i$  has a value of 0 or 1.

In the octal number system an arbitrary number is

$$N = a_{n-1}8^{n-1} + a_{n-2}8^{n-2} + \dots + a_08^0 + a_{-1}8^{-1} + \dots + a_{-m}8^{-m} \quad (1.4)$$

i.e.

$$N = \sum_{i=-m}^n a_i 8^i$$

where  $a_i$  has values of 0, 1, 2, 3, 4, 5, 6, 7.

The hexadecimal number system is in base  $p = 16$ , therefore, the value of each digit position may vary from 0 to 15.

In the hexadecimal number system the number  $N$  is in the form

$$N = a_{n-1}16^{n-1} + a_{n-2}16^{n-2} + \dots + a_016^0 + a_{-1}16^{-1} + \dots + a_{-m}16^{-m} \quad (1.5)$$

i.e.

$$N = \sum_{i=-m}^{n-1} a_i 16^i$$

where  $a_i$  has values of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Zero through 9 are used for the first ten, and A through F are used for the remaining six digits: 10 — A, 11 — B, 12 — C, 13 — D, 14 — E, 15 — F.

The major merit of the hexadecimal number system is that it is a convenient form of shorthand notation of binary numbers. This is fairly important, since the digit capacity range of numbers, instructions, and special binary words customarily used by the ES EVM

has become far wider. For example, the addresses of the main memory include 24 bits; special words depicting the status of the computation process have 64 bits. That is why, the possibility of writing large binary numbers in short became one of the causes of utilizing the hexadecimal system of numeration. More than that, the basic unit of information employed by the ES EVM is an eight-bit binary unit of code called *byte* and variables of number and instruction digit capacity are set as a multiple of bytes. Since any group of four bits may be converted directly to one hexadecimal digit, the binary unit of code (eight bits) can be conveniently converted into two hexadecimal digits. Decimal numbers may be also fairly well converted into hexadecimal. The use of the hexadecimal system has proved to be convenient for implementing four- and eight-digit accelerated shifts in computers, which adds to the acceleration of arithmetic and logic operations containing shifts.

**Example.** Write expanded numerals for the numbers  $657.341_8$  and  $BC.9A_{16}$ .

$$657.341_8 = 6 \cdot 8^2 + 5 \cdot 8^1 + 7 \cdot 8^0 + 3 \cdot 8^{-1} + 4 \cdot 8^{-2} + 1 \cdot 8^{-3}$$

$$BC.9A_{16} = 11 \cdot 16^1 + 12 \cdot 16^0 + 9 \cdot 16^{-1} + 10 \cdot 16^{-2}$$

## 1.2. Binary Coding of Decimal Numerals

The number system employed by a computer to perform arithmetic and logic operations on source numbers (operands) is called the basic system. For the digital computers this is the binary system of numeration.

In programming the programmer usually utilizes the decimal system, while for the entry in the computer the source information and instruction codes have to be converted from decimal to binary. The conversion from decimal to binary is carried out in two steps. First, the number is converted (encoded) into the Binary-Coded-Decimal (BCD) form. Then, the computer changes (decodes) from the binary-coded-decimal to binary. The ES EVM employs a binary-coded system based on the method of direct substitution (the 8421 code), when converted to binary are only digits, each separately, rather than the number. In practice, this conversion is carried out by the punchers. Therefore, the binary-coded decimal system is a number system in base 10 whose digits are encoded in the form of four bits (tetrads).

Conversion of decimal numbers into binary-coded-decimal notation is not difficult, as one has to remember only the binary codes for the ten decimal characters. These are

$$\begin{array}{l} 0 - 0000, \quad 1 - 0001, \quad 2 - 0010, \quad 3 - 0011, \quad 4 - 0100, \quad 5 - 0101, \\ 6 - 0110, \quad 7 - 0111, \quad 8 - 1000, \quad 9 - 1001 \end{array}$$

**Example.** Convert  $208.793_{10}$  to BCD.

Represent each numeral of the decimal number in the binary code:

2	0	8.	7	9	3
0010	0000	1000.	0111	1001	0011

Now, write the number in the BCD

$$208.793_{10} = 001000001000.011110010011_{2-10(\text{BCD})}$$

The reverse procedure to convert from BCD to binary is neither difficult. In order to carry out such a conversion the binary-coded-decimal number should be divided into tetrads on the left and right of the point. This done, convert the tetrads one-by-one from binary to decimal.

**Example.** Convert number  $100001010011.10010100$  from BCD to decimal.

1000	0101	0011.	1001	0100
8	5	3	9	4

$$100001010011.10010100_{(\text{BCD})} = 853.94_{10}$$

Even more simple are the rules for converting integer and fractional numbers from binary to octal and hexadecimal and vice versa.

To convert a binary number to octal, the binary number must be divided into groups of three bits (triads) from right to left for the integer part and from left to right for the fractional part, zeros being added to fill the extreme triads. Next, each triad is converted separately from binary to octal. The resultant sequence of octal digits will give us the octal form of the binary number being converted.

In converting from octal to binary three bits are substituted for each octal digit. The obtained sequence of bits will give us the binary representation of the octal number being converted.

The procedure used to convert from binary to hexadecimal is similar to that used for converting from binary to octal except that the binary number is broken into groups of four bits (tetrads) instead of groups of three (as in conversion to octal).

**Example.** Convert  $110110110.1101_2$  from binary to octal (1) and hexadecimal (2).

1. Break the binary number into triads and substitute octal digits for each group (triad), as follows:

110	110	110.	110	100
6	6	6.	6	4

$$110110110.1101_2 = 666.64_8$$

2. Separate the binary number into groups of four bits (tetrads) and substitute a hexadecimal digit for each group of four, as follows:

0001	1011	0110.	1101
1	B	6	D

$$110110110.1101_2 = 1B6.D_{16}$$

**Exercise**

Convert the binary numbers below to octal and hexadecimal:

- |                                     |  |
|-------------------------------------|--|
| 1. 1010100111.01110001 <sub>2</sub> | 11. 11010111.11010101 <sub>2</sub>     |
| 2. 1000110110.1111011 <sub>2</sub>  | 12. 11101100.10111101 <sub>2</sub>     |
| 3. 111100001.1000011 <sub>2</sub>   | 13. 11111101.11001110 <sub>2</sub>     |
| 4. 11001100.10001101 <sub>2</sub>   | 14. 10000001.00110011 <sub>2</sub>     |
| 5. 10000011.01010101 <sub>2</sub>   | 15. 10010010.01001001 <sub>2</sub>     |
| 6. 11111000.0001111 <sub>2</sub>    | 16. 10100100.10001011 <sub>2</sub>     |
| 7. 111000111.000111 <sub>2</sub>    | 17. 10110101.10010111 <sub>2</sub>     |
| 8. 100111000.1110101 <sub>2</sub>   | 18. 11001011.00111111 <sub>2</sub>     |
| 9. 11110000.01111001 <sub>2</sub>   | 19. 1110001101.1100011001 <sub>2</sub> |
| 10. 10000111.10000110 <sub>2</sub>  | 20. 100011111.01100110011 <sub>2</sub> |

**1.3. Representation of Numbers in Computer**

In modern computers all numbers can be represented in any one of two forms: natural known as fixed-point, or semilogarithmic (called floating-point).

In the *fixed-point (natural)* representation, the position of the point separating the integral part from the fractional part is fixed, i.e. the exponent of the numbers the computer deals with remains unchanged. At present, the fixed-point computers deal with number modulo  $<1$ . Therefore, if the word length (size) of a computer has  $n$  digit places, then the numbers represented in such a computer will lie within the range

$$2^{-(n-1)} \leq X \leq 1 - 2^{-(n-1)}$$

where  $2^{-(n-1)} = |X_{\min}|$  is the minimum number, and  $1 - 2^{-(n-1)} = |X_{\max}|$  is the maximum number.

Note that in this event one bit gives the sign of the number. The result of operations on numbers may be less than the least number, i.e. less than  $\underbrace{0.00 \dots 01}_{n-1} = 2^{-(n-1)}$  which corresponds to the com-

puter zero. If the result turns out to be greater than the largest number, i.e. greater than  $\underbrace{0.11 \dots 1}_{n-1} = 1 - 2^{-(n-1)}$ , then the so-

called computer word length overflow takes place. Without special corrections, these results cannot be used in further computations, and use should be made of correction factors in the form of scale factors.

Besides, when using the fixed-point form, all numbers are represented in the computer with different relative errors depending upon the number value. This is because the absolute error of the numbers represented in the computer word size (length) having  $n$  digit posi-

## PREFACE

In a short quarter-century, the electronic computer has had an enormous impact upon business, industry, science, education, and society in general. Scarcely an occupation or academic discipline has not been profoundly affected by the computer's speed and tireless capacity for work. The advent of large-scale integrated circuits, microprocessors, and large-capacity semiconductor memories have made possible powerful computer systems, cutting their cost, dimensions, and power consumption.

In order for automation to be introduced throughout the economy, a common engineering base is essential. The CMEA countries, therefore, have started mass production of a Unified System of Computers, the Cyrillic ES EVM title yielding the abbreviation which we shall now use.

The six base models differ in their specifications (speed, main storage capacity, etc.) and some 140 different peripherals are available. These peripherals substantially enhance the abilities of the ES EVM computers without affecting their software compatibility.

The ES EVM computer system has become a vital tool for any research worker who must solve complicated problems. They have been applied in the most diverse fields of science and technology, such as physics, chemistry, geology, biology, nuclear power engineering, space research, rocket technology, aviation, machine-tool industry, and management. There is hardly any industry which does not use or cannot use computers in some way.

The software for the ES range includes operating systems, automatic programming aids in standard languages and compilers.

Algorithmic languages are the most important part of the ES EVM software, and provide tools for solving problems on ES computers.

All those attending the computers, system programmers and operators, those involved in data processing, and computer maintenance specialists must be conversant with programming in such problem-oriented languages as ALGOL, FORTRAN, COBOL, Assembler, and PL/I.

This book is aimed at training such specialists of medium level. We believe this text will be of help to those using an independent approach to the study of computer programming and algorithmic languages.

## PART 1

# Fundamentals of Programming

## CHAPTER 1

### COMPUTER ARITHMETIC AND LOGIC CONCEPTS

#### 1.1. Number Systems

The *number (numeration) system* is a set of techniques and rules for representing numbers with the help of which a unique relationship can be established between any number and its representation in the form of a set of a finite number of symbols. The set of symbols used for this representation is known as *digits*.

The history of counting knows many diverse systems of numeration. However, an analysis of various number systems allows all the systems to be classified into two groups, positional and nonpositional, according to whether a digit value depends upon its location with respect to other digits.

*Nonpositional* are number systems in which the value of a given symbol never changes, regardless of its place or position in a given number. The known nonpositional systems of numeration employ either the addition principle (such systems are called additive systems), or the multiplication principle (such systems are known as multiplicative ones). The most simple is a number system consisting of one digit, a 1. An example is a sequence of strokes *////////*, in which the quantitative equivalent (weight) of each stroke (character) corresponds to one, and which represents a numeral for eight. Systems like this one are not used nowadays.

The Roman number system is a more intricate not 'place value' system which utilizes principles both of addition and subtraction. If a numeral of less quantitative equivalent appears after a numeral having a greater quantitative equivalent, their quantitative equivalents (weights) are added. If it appears before a greater numeral, their weights are subtracted. For example, in the Roman numeral IV, I means 1 and V means 5. When I appears before V, the number is 4 ( $5 - 1$ ). For the numeral VI, I still has a value of 1, and V a value of 5; but the number has a value of 6 ( $5 + 1$ ) because of the position of the numeral I. However, V always means 5. Nowadays, the Roman system of numeration is not used in the computation practice. It finds applications in various designations, such as numbering book chapters, marking hours in the face of clocks and watches,

etc. The not 'place value' (nonpositional) systems feature very intricate and cumbersome algorithms of number representation and execution of arithmetical operations of addition, subtraction, multiplication, and division, for which reason they are not used in the digital computers.

Used in the digital computers are *positional* ('place value') number systems. These are positional representation systems in which digits have weights which are multiplier values that depend upon their digit position in relation to the other digits of a number. In such systems as these, the weight (value) of each digit depends upon its position in the number representation. For example, in the number 575 the digit 5 is encountered twice. Its weight, however, is not the same in both cases: the least significant digit 5 occupies the units position and has a value of 5. The most significant digit 5 is in the hundreds position and stands for 500.

The most ancient of the known positional systems is the sexadecimal Babylonian system of numeration in which there were only two symbols used to represent one and ten. These two symbols were used repeatedly to write numerals from 1 to 59. Traces of this system remain till now. Partially it is used in time counting (an hour contains 60 min, a minute comprises 60 seconds), and in degree measurement of angles ( $1^\circ = 60'$ ,  $1' = 60''$ ). In addition to the Babylonian system, we know a base-20 system of numeration which was used by American Indian peoples, and is best known in the well-developed Mayan number system. Celtic traces of a base 20 were found in the French monetary system in which the basic unit of money, a franc, was a 20-sou unit.

Our present number system that has found most wide use, is also positional. Compared with the nonpositional systems, the major advantage of the positional number systems is in ease of number representation and simplicity of arithmetic operations.

An analysis of the decimal number system shows us that this system employs the principles of multiplication and addition. For example, consider the number 456. The '4' on the left means 4 hundreds and may be written as  $4 \times 10 \times 10$  (since  $100 = 10 \times 10$ ). The '5', the second digit from the left, means 5 tens and may be written as  $5 \times 10$ , and the '6', the third digit from the left, means 6 unities, and may be written as  $6 \times 1$ . Thus 456 can be represented in the form of the following expanded numeral (polynomial):  $456 = 4 \cdot 100 + 5 \cdot 10 + 6 \cdot 1$ , or  $456 = 4 \cdot 10^2 + 5 \cdot 10^1 + 6 \cdot 10^0$ .

All the positional systems of numeration feature a certain number known as the *base* of the number system. This is the amount by which a digit is multiplied or divided when moved to an adjacent digit position. In the decimal system, for example, a '3' becomes '30' if moved one digit position to the left, and '0.3' if moved a digit position to the right.

In any positional number system with a base  $p|p| \geq 1$ , the base of this system does not change whether we use ten or four or any number as a base. For example, the numeral '10' usually means 1 ten, when the base is 10, but if the base were eight, it would mean 1 eight, and if the base were two, it would mean 1 two, and so on. Remember that the symbol '10' always represents the base, since '1' in the next-to-last place indicates the base taken once. Further, we shall consider solely positional number systems employing the principles of multiplication and addition. In such systems with base  $p$ , an arbitrary number  $N$  will take the form

$$N = a_{n-1}p^{n-1} + \dots + a_1p^1 + a_0p^0 + a_{-1}p^{-1} + \dots + a_{-m}p^{-m} \quad (1.1)$$

or

$$N = \sum_{i=-m}^{n-1} a_i p^i$$

where  $p$  is a number system base representing an integer greater than 1 in absolute value,  $a_i$  are digits from the set  $0, 1, 2, \dots, p-1$ ,  $m$  is the number of digit places to the right of the point (the fractional part of the number), and  $n$  is the number of digit places to the left of the point (the integral part of the number).

The generally used decimal system of numeration fully satisfies the relation (1.1), i.e. an arbitrary number  $N$  in the decimal number system can be represented as follows

$$N = a_{n-1}10^{n-1} + a_{n-2}10^{n-2} + \dots + a_010^0 + a_{-1}10^{-1} + \dots + a_{-m}10^{-m} \quad (1.2)$$

or

$$N = \sum_{i=-m}^{n-1} a_i 10^i$$

Therefore, the generally used number representation is nothing more, than an enumeration of the coefficients of an expanded numeral (polynomial) shown in (1.1).

In writing a number, the number system is designated by putting the number base as a subscript which represents the base of the corresponding number system, the base being written in the decimal system. For example,  $763_8$  means that the number is in the octal numbering system.

**Example:** Write expanded numerals for the following numbers  $789.35_{10}$  and  $1101.0101_2$ .  $N = 789.35_{10} = 7 \cdot 10^2 + 8 \cdot 10^1 + 9 \cdot 10^0 + 3 \cdot 10^{-1} + 5 \cdot 10^{-2}$ ;  $N = 1101.0101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4}$ .

In the construction of computers and development of programming methods, use is made of the positional (place value) number systems with base 2, 8 and 16.

In the binary number system, an arbitrary number  $N$  will be as follows:

$$N = a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \dots + a_02^0 + a_{-1}2^{-1} + \dots + a_{-m}2^{-m} \quad (1.3)$$

i.e.

$$N = \sum_{i=-m}^{n-1} a_i 2^i$$

where  $a_i$  has a value of 0 or 1.

In the octal number system an arbitrary number is

$$N = a_{n-1}8^{n-1} + a_{n-2}8^{n-2} + \dots + a_08^0 + a_{-1}8^{-1} + \dots + a_{-m}8^{-m} \quad (1.4)$$

i.e.

$$N = \sum_{i=-m}^n a_i 8^i$$

where  $a_i$  has values of 0, 1, 2, 3, 4, 5, 6, 7.

The hexadecimal number system is in base  $p = 16$ , therefore, the value of each digit position may vary from 0 to 15.

In the hexadecimal number system the number  $N$  is in the form

$$N = a_{n-1}16^{n-1} + a_{n-2}16^{n-2} + \dots + a_016^0 + a_{-1}16^{-1} + \dots + a_{-m}16^{-m} \quad (1.5)$$

i.e.

$$N = \sum_{i=-m}^{n-1} a_i 16^i$$

where  $a_i$  has values of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Zero through 9 are used for the first ten, and A through F are used for the remaining six digits: 10 — A, 11 — B, 12 — C, 13 — D, 14 — E, 15 — F.

The major merit of the hexadecimal number system is that it is a convenient form of shorthand notation of binary numbers. This is fairly important, since the digit capacity range of numbers, instructions, and special binary words customarily used by the ES EVM

has become far wider. For example, the addresses of the main memory include 24 bits; special words depicting the status of the computation process have 64 bits. That is why, the possibility of writing large binary numbers in short became one of the causes of utilizing the hexadecimal system of numeration. More than that, the basic unit of information employed by the ES EVM is an eight-bit binary unit of code called *byte* and variables of number and instruction digit capacity are set as a multiple of bytes. Since any group of four bits may be converted directly to one hexadecimal digit, the binary unit of code (eight bits) can be conveniently converted into two hexadecimal digits. Decimal numbers may be also fairly well converted into hexadecimal. The use of the hexadecimal system has proved to be convenient for implementing four- and eight-digit accelerated shifts in computers, which adds to the acceleration of arithmetic and logic operations containing shifts.

**Example.** Write expanded numerals for the numbers  $657.341_8$  and  $BC.9A_{16}$ .

$$657.341_8 = 6 \cdot 8^2 + 5 \cdot 8^1 + 7 \cdot 8^0 + 3 \cdot 8^{-1} + 4 \cdot 8^{-2} + 1 \cdot 8^{-3}$$

$$BC.9A_{16} = 11 \cdot 16^1 + 12 \cdot 16^0 + 9 \cdot 16^{-1} + 10 \cdot 16^{-2}$$

## 1.2. Binary Coding of Decimal Numerals

The number system employed by a computer to perform arithmetic and logic operations on source numbers (operands) is called the basic system. For the digital computers this is the binary system of numeration.

In programming the programmer usually utilizes the decimal system, while for the entry in the computer the source information and instruction codes have to be converted from decimal to binary. The conversion from decimal to binary is carried out in two steps. First, the number is converted (encoded) into the Binary-Coded-Decimal (BCD) form. Then, the computer changes (decodes) from the binary-coded-decimal to binary. The ES EVM employs a binary-coded system based on the method of direct substitution (the 8421 code), when converted to binary are only digits, each separately, rather than the number. In practice, this conversion is carried out by the punchers. Therefore, the binary-coded decimal system is a number system in base 10 whose digits are encoded in the form of four bits (tetrads).

Conversion of decimal numbers into binary-coded-decimal notation is not difficult, as one has to remember only the binary codes for the ten decimal characters. These are

$$\begin{array}{l} 0 - 0000, \quad 1 - 0001, \quad 2 - 0010, \quad 3 - 0011, \quad 4 - 0100, \quad 5 - 0101, \\ 6 - 0110, \quad 7 - 0111, \quad 8 - 1000, \quad 9 - 1001 \end{array}$$

**Example.** Convert  $208.793_{10}$  to BCD.

Represent each numeral of the decimal number in the binary code:

2	0	8.	7	9	3
0010	0000	1000.	0111	1001	0011

Now, write the number in the BCD

$$208.793_{10} = 001000001000.011110010011_{2-10(\text{BCD})}$$

The reverse procedure to convert from BCD to binary is neither difficult. In order to carry out such a conversion the binary-coded-decimal number should be divided into tetrads on the left and right of the point. This done, convert the tetrads one-by-one from binary to decimal.

**Example.** Convert number  $100001010011.10010100$  from BCD to decimal.

1000	0101	0011.	1001	0100
8	5	3	9	4

$$100001010011.10010100_{(\text{BCD})} = 853.94_{10}$$

Even more simple are the rules for converting integer and fractional numbers from binary to octal and hexadecimal and vice versa.

To convert a binary number to octal, the binary number must be divided into groups of three bits (triads) from right to left for the integer part and from left to right for the fractional part, zeros being added to fill the extreme triads. Next, each triad is converted separately from binary to octal. The resultant sequence of octal digits will give us the octal form of the binary number being converted.

In converting from octal to binary three bits are substituted for each octal digit. The obtained sequence of bits will give us the binary representation of the octal number being converted.

The procedure used to convert from binary to hexadecimal is similar to that used for converting from binary to octal except that the binary number is broken into groups of four bits (tetrads) instead of groups of three (as in conversion to octal).

**Example.** Convert  $110110110.1101_2$  from binary to octal (1) and hexadecimal (2).

1. Break the binary number into triads and substitute octal digits for each group (triad), as follows:

110	110	110.	110	100
6	6	6.	6	4

$$110110110.1101_2 = 666.64_8$$

2. Separate the binary number into groups of four bits (tetrads) and substitute a hexadecimal digit for each group of four, as follows:

0001	1011	0110.	1101
1	B	6	D

$$110110110.1101_2 = 1B6.D_{16}$$

**Exercise**

Convert the binary numbers below to octal and hexadecimal:

- |                                     |  |
|-------------------------------------|--|
| 1. 1010100111.01110001 <sub>2</sub> | 11. 11010111.11010101 <sub>2</sub>     |
| 2. 1000110110.1111011 <sub>2</sub>  | 12. 11101100.10111101 <sub>2</sub>     |
| 3. 111100001.1000011 <sub>2</sub>   | 13. 11111101.11001110 <sub>2</sub>     |
| 4. 11001100.10001101 <sub>2</sub>   | 14. 10000001.00110011 <sub>2</sub>     |
| 5. 10000011.01010101 <sub>2</sub>   | 15. 10010010.01001001 <sub>2</sub>     |
| 6. 11111000.0001111 <sub>2</sub>    | 16. 10100100.10001011 <sub>2</sub>     |
| 7. 111000111.000111 <sub>2</sub>    | 17. 10110101.10010111 <sub>2</sub>     |
| 8. 100111000.1110101 <sub>2</sub>   | 18. 11001011.00111111 <sub>2</sub>     |
| 9. 11110000.01111001 <sub>2</sub>   | 19. 1110001101.1100011001 <sub>2</sub> |
| 10. 10000111.10000110 <sub>2</sub>  | 20. 100011111.01100110011 <sub>2</sub> |

**1.3. Representation of Numbers in Computer**

In modern computers all numbers can be represented in any one of two forms: natural known as fixed-point, or semilogarithmic (called floating-point).

In the *fixed-point (natural)* representation, the position of the point separating the integral part from the fractional part is fixed, i.e. the exponent of the numbers the computer deals with remains unchanged. At present, the fixed-point computers deal with number modulo  $<1$ . Therefore, if the word length (size) of a computer has  $n$  digit places, then the numbers represented in such a computer will lie within the range

$$2^{-(n-1)} \leq X \leq 1 - 2^{-(n-1)}$$

where  $2^{-(n-1)} = |X_{\min}|$  is the minimum number, and  $1 - 2^{-(n-1)} |X_{\max}|$  is the maximum number.

Note that in this event one bit gives the sign of the number. The result of operations on numbers may be less than the least number, i.e. less than  $0.00 \dots 01 = 2^{-(n-1)}$  which corresponds to the com-

puter zero. If the result turns out to be greater than the largest number, i.e. greater than  $0.11 \dots 1 = 1 - 2^{-(n-1)}$ , then the so-

called computer word length overflow takes place. Without special corrections, these results cannot be used in further computations, and use should be made of correction factors in the form of scale factors.

Besides, when using the fixed-point form, all numbers are represented in the computer with different relative errors depending upon the number value. This is because the absolute error of the numbers represented in the computer word size (length) having  $n$  digit posi-

tions equals  $\delta_{abs} = \underbrace{0.00 \dots 01}_{n-1}$ , i.e. the least significant digit,

while the relative error is computed as the ratio of the absolute error to the modulus of the number being written, i.e.  $\delta_{rel}$  varies from  $\delta_{abs}/N_{min}$  to  $\delta_{abs}/N_{max}$ .

The other form of representing numbers in a computer is the *floating-point (semilogarithmic)* form. If in the fixed-point form encoded in the word size is solely the number mantissa, since the exponent has been fixed, in the floating-point form both the mantissa and exponent are encoded. In the place-value (positional) systems of numeration with base  $p$ , numbers can be represented as  $X = Yp^m$ , where  $Y$  is the mantissa of the number, and  $m$  is the exponent of the number. Encoding separately the exponent and mantissa gives us the floating-point form of the number. This form of number encoding, however, produces ununique representation. Indeed, number  $X = 57.16_{10}$  may be written as  $X = 57.16 \cdot 10^0$ ,  $X = 0.5716 \cdot 10^2$ ,  $X = 5716 \cdot 10^{-2}$ , etc.

In order to provide the unambiguity of floating-point representation of numbers, the following limitations to writing the mantissa should be used: the mantissa should be modulo less than 1 and the first digit in the mantissa should be significant.

With the binary system of numeration these limitations can be written in the form

$$1/2 \leq |Y| < 1 \quad (1.6)$$

The floating-point representation of numbers which satisfies the condition (1.6) is called the *normal semilogarithmic form*, and the numbers represented in this form are known as *normalized*. Let us define the range of numbers represented as normalized. The word size is selected the same as for the fixed-point representation, i.e. of  $n$  digit places. Allocate  $m$  digit places for writing the mantissa and  $n - m$  places for the exponent. If that is the case, the maximum number

$$|X_{max}| = 2^{+(n-m-1)} \underbrace{0.11 \dots 1}_{m-1},$$

$$\text{i.e. } |X_{max}| = 2^{+(n-m-1)} (1 - 2^{-(m-1)})$$

The minimum value of the number being represented will be

$$|X_{min}| = 2^{-(n-m-1)} \underbrace{0.10 \dots 0}_{m-1},$$

$$\text{i.e. } |X_{min}| = 2^{-(n-m-1)} 2^{-1} = 2^{m-n+1-1} = 2^{m-n}$$

Therefore, the range of number representation in the floating-point form is as follows:

$$2^{m-n} \leq |X| \leq 2^{+(n-m-1)} (1 - 2^{-(m-1)})$$

Determine errors occurring in the floating-point representation of numbers. The absolute error of representation does not exceed the least significant digit of the mantissa, i.e.  $\delta_{abs} = 2^{-(m-1)}$ .

The relative error of numbers represented in the normal floating-point form is as follows:

$$\delta_{rel} = \delta_{abs}/Y = 2^{-(m-1)} \left( \frac{1}{2} \text{ to } 1 \right) \approx 2^{-m} \text{ to } 2^{-(m-1)}$$

It is obvious that the accuracy of number representation in the floating-point form is independent of the number value.

Let us compare the two forms of representing numbers in a computer by some characteristics.

**Range of numbers represented.** It follows from the above calculations that the range of fixed-point numbers that can be represented in computers is far less than that of floating-point numbers, the word size being the same.

**Error of number representation.** Because in the fixed-point computers the relative error is a function of the absolute value of the number, the accuracy of representing small numbers will be low, whereas the accuracy of number representation in the floating-point computers remains the same regardless of the value of the number.

**Preparation of source data.** When the source data are prepared for the input into a fixed-point computer, use should be made of scale factors, which makes more complicated the preparation of a problem for the entry in a computer and program debugging. Possible overflow of the word size or occurrence of a computer zero necessitates the correction of the scale factors. The floating-point computers are free from these disadvantages.

**Computation speed and hardware.** In the course of floating-point number representation, the computer separately processes the mantissa and exponent, performs additional operations of exponent matching and mantissa unnormalizing. Therefore, with the floating-point computers the speed of processing operations is less than with the fixed-point computers, and their processor design is more sophisticated.

**Usability in computers.** Both forms of number representation have found their applications in the ES EVM computers. They are used as dictated by the type of the problem under solution, with a view to making the most of their advantages.

#### **1.4. Representing Negative Numbers in Digital Computers.**

##### **True Form, Ones-Complement Form, and Twos-Complement Form. Modified Codes**

So far we have considered only positive numbers; but some means must be provided for indicating when a number is negative. Since there are only two signs, '+' and '-', only one bit is needed to di-

stinguish between them and normally the left-hand bit in the word is used for the purpose. In the fixed-point computers the number sign is assigned a certain number of places found to the left of the most significant bit of the number. In the floating-point computers the signs of the exponent and mantissa are represented separately. This is done as follows: if the places allocated for the sign are filled with ones, the number is negative, if with zeros, the number is positive.

If two binary positions are allocated for the sign, the resultant code is known as the modified code.

Therefore, the sign bit function is as follows

$$f(x) = \begin{cases} 0, & \text{if } X \geq 0 \\ 1, & \text{if } X < 0 \end{cases} \quad (1.7)$$

To determine the sign of the result of such arithmetic operations as multiplication and division, use is made of the following operations

$$f(x \cdot y) = f(x) \oplus f(y) \quad \text{and} \quad f(x/y) = f(x) \oplus f(y) \quad (1.8)$$

In formulas (1.8) the addition is modulo 2, i.e. the addition is bit-by-bit according to the binary arithmetic rules, and a carry one, if appears, is dropped. For example, if two places are assigned for the sign, then the realization of formulas (1.8) will take the form:

1. Both numbers are positive  $00 \oplus 00 = 00$  } the result is
2. Both numbers are negative  $11 \oplus 11 = 00$  } positive
3. One number is positive, the other is negative:  $00 + 11 = 11$  or  $11 + 00 = 11$ , the result is negative.

To represent signed numbers there are three forms used in the modern computers. These are true, ones complement and twos complement forms. The most simple is the true form (also true complement, radix complement). Let this be conventional representation in the natural form with the sign represented in the sign places in compliance with formula (1.7).

The *true form* will be designated  $[X]_{tr}$ :

$$\begin{aligned} [X]_{tr} &= 00x_n x_{n-1} \dots x_0, & \text{if } X > 0 \\ [X]_{tr} &= 11x_n x_{n-1} \dots x_0, & \text{if } X < 0 \end{aligned}$$

**Example.** Represent binary numbers  $X_1 = +0.1011101_2$  and  $X_2 = -0.1011001_2$  in the true form. Reserve two bit places for the sign.

$$[X_1]_{tr} = 00 \ 1011101 \quad [X_2]_{tr} = 11 \ 1011001$$

A 0 in the true form has two designations—a plus 0 and a minus 0.

$$+0 = 00 \ 00 \dots 0, \quad -0 = 11 \ 00 \dots 0$$

The true form is used for representation of positive binary numbers during execution of operations in the computer processor, for storage of positive and negative numbers in the storage and input/output devices. During the execution of operations on negative numbers in the true form, one has to determine the sign of the result. To this end, the moduli of the numbers must be compared and the sign of the greater number should be assigned to the result. In a computer this would mean more control hardware and a lower algebraic addition speed. To replace the algebraic addition with arithmetic addition with the sign of the algebraic addition result determined automatically, use should be made of special number representation forms, the so-called ones and twos complement forms.

The *ones complement representation* is in compliance with the following relations:

$$[X]_{\text{ones}} = 00x_n x_{n-1} \dots x_0, \quad \text{if } X > 0$$

$$[X]_{\text{ones}} = 11\bar{x}_n \bar{x}_{n-1} \dots \bar{x}_0, \quad \text{if } X < 0$$

where  $\bar{x}_i = 0$  if  $x_i = 1$ , and  $\bar{x}_i = 1$  if  $x_i = 0$ .

In fact ones complementation of a negative binary number is equivalent to placing ones in the sign bit places and changing all the noughts to ones and all the ones to noughts. In the ones complement form a 0 has two representations

$$[+0]_{\text{ones}} = 00\,000 \dots 0 \quad \text{and} \quad [-0]_{\text{ones}} = 11\,111 \dots 1$$

**Example.** Represent the following binary numbers in the ones complement form

$x_1 = 0.10100_2$	$[X_1]_{\text{ones}} = 00\,10100$
$x_2 = -110.01011_2$	$[X_2]_{\text{ones}} = 11\,001.10100$
$x_3 = -11001011_2$	$[X_3]_{\text{ones}} = 11\,00110100$
$x_4 = 111001_2$	$[X_4]_{\text{ones}} = 00\,111001$

The *twos complement representation* is in accordance with the following relations:

$$[X]_{\text{twos}} = 00x_n x_{n-1} \dots x_0, \quad \text{if } X \geq 0$$

$$[X]_{\text{twos}} = 11\tilde{x}_n \tilde{x}_{n-1} \dots \tilde{x}_0, \quad \text{if } X < 0$$

where  $\tilde{x}_n, \tilde{x}_{n-1}, \dots, \tilde{x}_0$  is 1's complement of  $|X|$  ( $X$  is a binary fraction).

In compliance with this to obtain the twos complement form of negative binary number, the given number must be represented in the ones complement form and the number  $0.0 \dots 01$ , i.e. the least significant bit (LSB), should be added to it.

To perform reconversion, a number comprising only ones, including the sign bits (MSB carry being dropped) must be added to the

twos complement representation of the negative number, and the result of addition must be inverted, i.e. all the ones must be changed to noughts, and all the noughts, to ones, leaving the sign bits unchanged.

**Example.** Represent binary numbers  $X_1$  and  $X_2$  in the twos complement form.

$$\begin{aligned} X_1 &= 0.10100_2 & [X_1]_{twos} &= 00\ 10100 \\ X_2 &= -110.01011_2 & [X_2]_{twos} &= [X_2]_{ones} + 0.00000001 \\ & & [X_2]_{ones} &= 11\ 001.10100 \\ [X_2]_{twos} &= 11\ 00110100 + 0.00000001 = 11\ 00110101 \end{aligned}$$

For convenience of conversion the number  $X_2$  has been represented as a binary fraction in the fixed-point form.

**Example.** Convert from twos complement to true form. Use the result of the above example conversion.

$$\begin{aligned} [X_2]_{twos} &= 11\ 00110101 & 11\ 00110101 \\ & & +11\ 11111111 \\ & & 11\ 00110100 \\ [X_2]_{tr} &= 11\ 11001011 \end{aligned}$$

Note, that a 0 in the twos complement form has only one representation  $[0]_{twos} = 00\ 00 \dots 0$ .

For representation of floating-point numbers in the true, ones- and twos-complement forms, the exponent and mantissa must be represented separately.

**Example.** Represent number  $X = -1101.0011$  in the true, ones- and twos-complement forms.

Represent number  $X$  in the floating-point (semilogarithmic) form:  $X = -0.11010011_2 \cdot 2^{100}$ . Represent the number in the true, ones- and twos-complement forms:

$$\begin{aligned} X_{tr} &= 11\ 11010011\ 00\ 100 \\ X_{ones} &= 11\ 00101100\ 00\ 100 \\ X_{twos} &= 11\ 00101101\ 00\ 100 \end{aligned}$$

## CHAPTER 2

### COMPUTER ORGANIZATION

#### 2.1. General Concept of the Third-Generation Computer

By now there have been engineered and are being devised many computers of diverse types and applications. These are both large universal computers often combined into computer systems, and

special computers knowingly deprived of certain capabilities, which feature high speeds of operation and are used to control automatic transfer lines, production processes, navigation systems of ships, etc.

In the CMEA countries quite a series of electronic computers have been recently developed which are known under the name the Unified System of Electronic Computers (Edinaya Sistema EVM — ES EVM). The engineering development of the ES EVM is underlain by the following principles.

1. **Program compatibility.** This means that a package of applied programs can be used on any model of the ES EVM computers. The necessity and great utility of this feature are evident. This principle allows for standardization of programs and development of unified software.

2. **Information compatibility.** Implementation of this principle is based on the standardization of coding and information representation on external media and its similar acceptance by all computers of the ES EVM series. The same principle also covers the possibility of information writing and readout on peripheral devices of different computers.

3. **Multiprogramming.** This principle provides a method of computer operation in which two or more application programs are executed simultaneously by the interleaved allocation of a single set of computer resources. The multiprogramming techniques dramatically raise the computer efficiency owing to the concurrent use of the central and peripheral equipment.

4. **Use of an ES EVM computer as a base machine for various computer systems.** By this is meant the possibility of furnishing a computer with different numbers of peripheral devices, I/O channels, and the use of working storage capacities varying with the problems being solved on the machine. If that is the case, it is natural that for each operating system there is a certain minimized configuration which allows its normal functioning.

In what to follow, the organization of the computer and the construction and operation of its devices will be considered with the ES-1030 computer of the unified system ES EVM used as an example.

The ES-1030 is a universal computer of medium computational capabilities. It employs the unified set of instructions and software, standard interfaces, and peripherals of the ES EVM.

The typical configuration of the ES-1030 computer fully depicting the organization of the ES EVM series is shown in Fig. 2.1.

**Processor**, also a central processing unit (CPU), is the most essential part of a computer which decodes instructions and controls the hardware (electronic circuits) used to execute them. It consists of the control unit and arithmetic unit (or a single unit that performs both

functions) and, in most systems, main storage (memory). The primary functions of the CPU are as follows.

1. Performing of all arithmetic and logic operations (commands, instructions). Study of any computer is started with the study of the

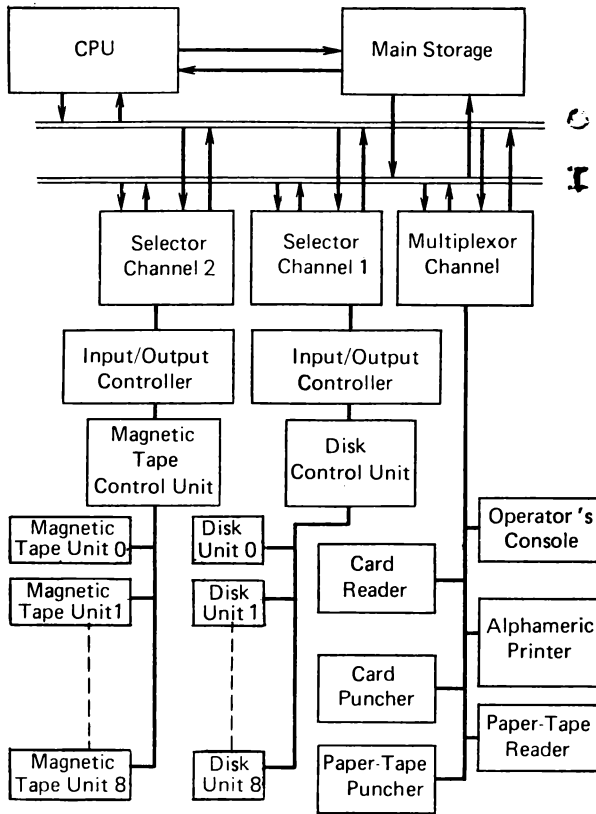


Fig. 2.1. Block diagram of computer

set of instructions implemented in it, and with the principles and algorithms of their execution. The instruction set helps to form an opinion of the computer itself, its complexity, universality, etc. All the computers of the ES EVM series have similar instruction sets, which accounts for their program compatibility.

2. Holding and accessing the order code of the computer. First of all it is the control of the sequence of the instruction execution in compliance with the program being handled and execution of the instructions themselves. The execution of even a most simple instruction involves a number of operations to be performed on the instru-

ction or information. Such operations may be various transfers, shifts, logic and arithmetic additions, condition checks, etc. Besides, the CPU organizes fetching the instructions and operands participating in the execution of the above-mentioned commands from the main storage. More than that, the CPU manages references to the input/output channels and handles interrupts from the channels caused by normal or abnormal terminations of input/output operations.

3. In addition to the interrupts (service requests) from the channels, the processor functions include acceptance of interrupts from the control circuits, clock (timer), program interrupts and control of the priority they are dealt with. All this together is known as an interrupt system which is a hardware facility of implementing the multiprogramming mode of problem handling.

Channels are hardware means used to control input/output operations. These devices appeared in the computer structure due to the following requirements imposed on the modern computers:

- Input/output operations must be performed concurrently with the operation of the central processor unit;
- There should exist the possibility of simultaneous execution of several input/output operations.

Let the imposing of these requirements be illustrated by an example. We have to enter a card data file into the computer memory. In the modern computers, the speed of operation of the computer main memory is 500 to 700 times that of a card reader. Therefore, in a computer designed according to the operating principle of second-generation one-program computers, the processor will have downtime 99 percent of the input time. Use of the channels, or input/output processors as they are sometimes called, allows control of data exchange between a peripheral device and main storage memory to be transferred to the channels, while the CPU is being switched to handle another program. More than that, the use of several channels makes it possible to exchange information simultaneously with several peripheral devices, while the central processing unit may again be busy with a parallel problem requiring its operation. Besides, input/output (I/O) devices essentially differ in their principles of operation, functions and, naturally, in the speed of operation. Therefore, a problem arises of more efficient use of the channels.

Such peripheral devices as disk drives (units) and tape drives (units) are high-speed devices and call for individual channel service. This means that the speed of operation of these peripheral devices is comparable to that of a channel and its throughput for which reason the channel turns out to be unable to service several tape and disk drives at the same time. Then, the data exchange with high-speed peripheral devices is under control of selector channels. However, there are peripheral devices of relatively low speed, such as card readers, card punches, paper tape readers and punches, printers, etc.

Such low-speed devices are unable to fully utilize the channel. To overcome this difficulty, a channel operating with low-speed I/O devices is designed with several subchannels each of which provides service to one of the peripheral (I/O) devices. The channel is able to deal with all the subchannels and to control data exchange in them by scanning them one after another, so that each of the low-speed I/O devices operates, as if it were the only one serviced by the channel. The channel is thus said to operate in a multiplex, or a byte mode. Hence, its name is a multiplexor channel. So, the multiplexor channel operates in a multiplex mode, and the selector channel, in a burst mode. At the same time, the multiplexor channel may also operate in a burst mode, if a peripheral device has requested a burst mode service.

**The main storage**, also main memory, primary storage, and internal storage, is used for information receipt, storage and display. This information may differ both in the representation format and in its function. Let it be explained by an example as follows. We have a conventional desk calculator which in our example performs the function of an arithmetic unit, i.e. a unit directly performing a series of arithmetic operations. The function of a control unit is performed by the operator of the desk calculator. Prior to calculations, the operator writes down, in a sheet of paper, certain numbers on which the arithmetic operations will be performed. If these calculations are complicated, then you have to write down a formula by which the calculations will be made. After the calculations are completed, the used information and formulas may be erased, and new formulas and numbers can be written in the same place. If in changing over from some calculations to other certain intermediate information is to be saved, this information can also be written in the same sheet of paper. So, the main memory in a computer performs the function of such a sheet of paper. A counterpart of the calculation formula in our example is the program stored as well as the information in the main memory. The main memory capacity may vary with each model of computer. It is limited by the maximum address that can be accessed and constructional features of the computer.

**Input/output controller** (control unit) serves to control one or several peripherals and to interface the peripherals with a channel. As the peripheral devices differ from one another in the operating principle and information representation form, so do their controllers. However, each of the controllers receives information from the channel in a standard way. The same takes place in the back flow of information which is read by the peripheral device and converted by the input/output controller into a standard form. More than that, the control pulses (signals) transferred between the channel and I/O controller are standard either, their sequence and functions being the same, regardless of the I/O controller type. This also accounts

for the standardization of the cables connecting the I/O controller to the channels. The set of such standard cables and strictly regulated control pulses both from the channel and from the input/output controller is called a standard interface. Any peripheral device (if the controller is constructionally an integral part of it), or a separate I/O controller designed for the ES EVM computers can be connected to a channel with the aid of the standard interface. There are also available devices converting information sent over communication lines into a form suitable for the standard interface. All this allows the user of the computer either to modify or adapt the configuration and quantity of peripheral devices to the nature of the problems being solved.

Therefore, being acquainted with the block diagram of the third-generation computer, we may arrive at the conclusion that a modern computer is of an intricate structure, and it would be more correct to call it a computer system, inasmuch as its configuration can be changed within wide limits. Considered below in more detail are the central and a number of peripheral devices, much attention being given to the operating principles and options offered to the user, rather than to their actual circuits and constructional features.

## **2.2. Data Representation in the Main Storage**

In the previous chapter we have acquainted ourselves with the methods of representing decimal numbers in the binary, hexadecimal, and binary-coded decimal (BCD) systems of numeration.

Regardless of the coding system a number is represented in a computer in the form of a certain binary equivalent. This is because the computer is built on binary elements, i.e. it utilizes elements with only two different states which are labelled zero and one. Theoretically, one can imagine a computer operating directly in the decimal number system. However, to this end use should be made of logic and storage elements having ten fixed states and featuring high reliability and high speed of operation. There are no such elements as yet, and any information entered into the computer, either numeric or alphabetic, should be changed into some binary equivalent which will be manipulated and processed in the computer. In order to make the same information be accepted by all program-compatible computers in a similar way, the ES EVM computers employ a unified standard code DKOI (dvoichny kod otobrazheniya informatsii) similar to the Extended Binary Coded Decimal Interchange Code (EBCDIC) used by IBM. In it an eight-bit binary code corresponds to each character. Given below is a table of characters and their binary codes represented in the hexadecimal form (Table 2.1). To make use of table easy, the leftmost column lists binary equivalents (bits) corresponding to each hexadecimal character.

Table 2.1

Hexadecimal form	DKOI (EBCDIC) Code									
	DKOI	Hex	DKOI	Hex	DKOI	Hex	DKOI	Hex	DKOI	Hex
0—0000	A	C <sub>1</sub>	Q	D <sub>8</sub>	6	F <sub>6</sub>	+	4E	Б	42
1—0001	B	C <sub>2</sub>	R	D <sub>9</sub>	7	F <sub>7</sub>	—	6D	Г	44
2—0010	C	C <sub>3</sub>	S	E <sub>2</sub>	8	F <sub>8</sub>	>	6E	Д	45
3—0011	D	C <sub>4</sub>	T	E <sub>3</sub>	9	F <sub>9</sub>	<	4C	Ж	47
4—0100	E	C <sub>5</sub>	U	E <sub>4</sub>	/	61	(	4D	З	48
5—0101	F	C <sub>6</sub>	V	E <sub>5</sub>	&	50	)	5D	И	49
6—0110	G	C <sub>7</sub>	W	E <sub>6</sub>	blank	40	[	4A	Й	51
7—0111	H	C <sub>8</sub>	X	E <sub>7</sub>	—	60	]	6A	Ј	53
8—1000	I	C <sub>9</sub>	Y	E <sub>8</sub>	!	4F	□	5F	П	57
9—1001	J	D <sub>1</sub>	Z	E <sub>9</sub>	.	4B	=	7E	Ф	64
A—1010	K	D <sub>2</sub>	C	F <sub>0</sub>	,	6B	"	7F	Ц	66
B—1011	L	D <sub>3</sub>	1	F <sub>1</sub>	#	7B	;	5E	Ч	67
C—1100	M	D <sub>4</sub>	2	F <sub>2</sub>	*	5C	:	7A	Ш	68
D—1101	N	D <sub>5</sub>	3	F <sub>3</sub>	%	6C	?	6F	Щ	69
E—1110	O	D <sub>6</sub>	4	F <sub>4</sub>	@	7C	!	5A		
F—1111	P	D <sub>7</sub>	5	F <sub>5</sub>	\	7D				

In the ES EVM the smallest unit of information that can be processed and addressed in the main memory is a *byte* which is eight bits in length. In all conversions, transfers and transmission operations each byte of information is subjected to the parity check, i.e. it is checked to determine if there is an odd number of one bits. Therefore, in the code there is an additional (9th) parity or check bit used to make the total number of one bits odd. This kind of check is called 'modulo-2 check'.

### Example.

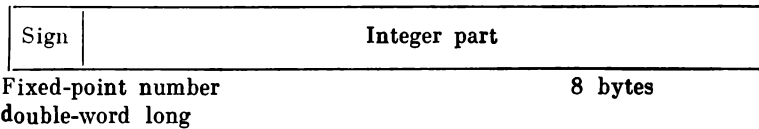
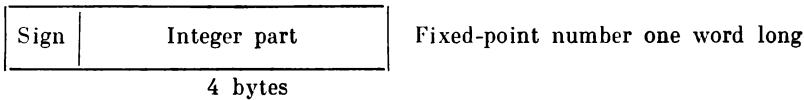
10110100 1	}	bytes with correct parity
11001011 0		
11101001 1	}	bytes with incorrect parity
01100101 0		

With the ES EVM, a fixed number of four bytes is called a *word*. Thus, a computer word in the ES EVM contains 32 information and 4 check bits. Certain instructions deal with information half-word (two bytes) or double-word (eight bytes) in length. All this applies to the operand-length oriented instructions which can manipulate only with fixed lengths. However, there is a series of instructions

which can deal with information whose length varies within the limits from one to 256 bytes.

Depending upon the instruction being executed, information (data) may be interpreted as fixed-point numbers, floating-point numbers (semilogarithmic form of data representation), decimal numbers and bit strings for logical operations.

By the *fixed-point number* is meant a signed binary integer which may be word or halfword in length. The scale of notation for the fixed-point numbers is selected so that the point is to the right of the least significant digit



The most significant leftmost zero bit represents the sign of the number. If the most significant bit (MSB) is a 0, the number stored has a positive value. If the MSB is a 1, the number has a negative value. Thus, the range of fixed-point numbers lies within the limits  $+2^{31} - 1 \geq X \geq -2^{31}$  ( $2^{31} = 2147483648$ ).

Note also that the largest negative number that can be represented in the form of a fixed-point binary number is greater than the largest positive number by a 1.

Let us consider some examples of representing numbers in a computer. For convenience, write fixed-point numbers with hexadecimal digits.

### Example.

<i>Decimal number</i>	<i>Fullword</i>	<i>Halfword</i>
67345	00010711	—
3104	00000C20	0C20
8	00000008	0008
0	00000000	0000
2147483647	7FFFFFFF	—
32767	00007FFF	7FFF

*Negative numbers are represented in the twos complement form*

—67345	FFFEF8EF	
—3104	FFFFF3E0	F3F0
—8	FFFFFFF8	—
—2147483647	80000001	—
—2147483648	80000000	—
—32767	FFFF8001	8001
—32768	FFFF8000	8000

Fixed-point halfwords are expanded to fullwords by repeating the sign bit of the halfword in each of the 16 leftmost bits, so that the change in the bit capacity does not lead to changes in the absolute value and sign of the number in question.

The *floating-point number* is used to represent fractions. A 32-bit is divided into three distinct parts to represent floating-point numbers. The leftmost bit is the sign of the fraction, with 0 representing positive and 1 negative. The next seven bits represent the exponent, or characteristic. The rightmost 24 bits represent the value of the fraction.

If in a certain actual calculation all the numbers involved were fixed-point numbers, then these numbers could be interpreted as integers only at the end of the calculation; keeping in mind the scale, we could determine the position of the point separating the integer part from the fractional one. However, in most calculations the position of the point may vary within fairly wide limits, and it is difficult to determine its probable position at the end of the calculations.

A floating-point number is represented in the ES EVM in compliance with the formula

$$Y = M \cdot 16^p$$

where  $Y$  is a floating-point number;  $M$  is the mantissa (sometimes the argument or coefficient);  $p$  is the exponent (sometimes characteristic) and 16 is the exponent base.

The floating-point number format is as follows

Sign	Characteristic	Mantissa
0 1	7 8	31

Sign	Characteristic	Mantissa
0 1	7 8	63

Thus, depending upon the format, the mantissa of a floating-point number is expressed as a six-digit hexadecimal number of single precision, or fourteen-digit hexadecimal number which corresponds to a floating-point double-precision number. By using double precision, we get twice as many significant digits, but the use of double-precision numbers takes more main storage space and takes more time than the corresponding single-precision operations.

The first specific feature of the floating-point number representation is that the number base is 16 rather than 2 as the case was with the previous computers. The use of this base has two advantages: each group of four bits beginning at the base point may be converted directly to one hexadecimal digit; it becomes possible to facilitate

the execution of operations containing shifts. Shifting is made four bits at once.

The second feature of representing floating-point binary numbers in the ES EVM is that the number exponents are represented increased by  $2^6 = 64$ . As a result, the negative and positive exponents lie within the limits

$$0 \leq p^* \leq 127 \text{ in place of } -64 \leq p \leq 63$$

An exponent shifted by 64 is called a 'characteristic'. Since all characteristics are positive, operations on them become simplified. Exponents can be compared with no analysis of their signs.

The mantissa of a floating-point number is its fractional part and should lie within the limits

$$1/16 \leq M < 1$$

In this event, the floating-point number is called normalized.

The decimal point should always precede the leftmost, most significant digit of the mantissa. Therefore, if a floating-point number is normalized, the first hexadecimal digit of the mantissa should be nonzero, and there should be no mantissa overflow condition. Operations on floating-point numbers may unnormalize the number either on the left or on the right. In the former case, a mantissa overflow condition arises as a result of operations performed by the circuits producing the termination condition codes. In the latter case, unnormalization is indicated by zero content in the most significant hexadecimal digit (or several digits) of the mantissa. In both cases, upon completion of the operation the number should be normalized in compliance with Table 2.2.

The characteristic of a floating-point number is represented as the exponent added to 64:

$$x = p + 64$$

where  $x$  is the characteristic and  $p$  is the exponent of a floating-point number.

Therefore, the floating-point number representation range is

$$16^{-65} \leq Y \leq (1 - 16^{-6}) 16^{63} \text{ for single-precision operations,}$$

and

$$16^{-65} \leq Y \leq (1 - 16^{-14}) 16^{63} \text{ for double-precision operations.}$$

In the decimal number system this range in both cases is as follows

$$5.4 \cdot 10^{-79} \leq Y \leq 7.2 \cdot 10^{75}$$

Both for positives and negatives the mantissa is expressed in the true form. The difference in signs affects the value of the leftmost

Table 2.2

Unnormalized type	Mantissa	Characteristic
Unnormalizing to the right	Leftward shift of the mantissa until the most significant hexadecimal digit of the mantissa becomes nonzero	Rightward shift of the characteristic as many places as hexadecimal digits the mantissa has been shifted for
Unnormalizing to the left	Shift of the mantissa one hexadecimal digit to the right	Leftward shift of the characteristic one digit place

zero digit which is the sign digit. The sign bit is 0 for positive numbers and 1 for negative numbers.

A number with a null characteristic, a null mantissa, and positive sign (i.e. a 0 in the sign place) is called a true zero.

The sign of the operation results is set according to the rules of algebra.

Several examples of representing floating-point numbers are as follows

<i>Decimal number</i>	<i>Fixed-point binary number</i>	<i>Floating-point binary number</i>
1	00000001	41100000
10	0000000A	41A00000
0	00000000	00000000 00000000
67 345	00010711	45107110
-67 345	FFFEF8EF	C5107110

Decimal numbers are represented in the form of right-justified signed integers in the true representation. The decimal digits from 0 to 9 have their corresponding binary codes from 0000 to 1001. Codes 1010 are inadmissible as codes of digits and are interpreted as codes of sign, codes 1010, 1100, 1110 and 1111 corresponding to the sign 'plus', and codes 1011 and 1101, to the sign 'minus'.

While the fixed- and floating-point binary numbers can be called onto certain registers of the processor accessible to the user, the decimal numbers which are operands of decimal arithmetic instructions, can be used only in the main storage. Fields they reside in can start with any byte and have a length from 1 to 16 bytes.

The sign of a decimal number is always represented in the right-most address high byte.

The ES EVM employs two formats of data representation. These are the zoned (unpacked) decimal and packed decimal forms.

In the *zoned format* decimal numbers usually get to the main storage as a result of data exchange with peripheral devices and have the following form:

Zone	Digit	Zone	Digit		Zone	Digit	Sign	Digit
Zero byte		1st byte			15th byte		16th byte	

In the zoned (unpacked) format the low-order four bits of each byte are numeric, standing for binary-coded decimal digits. The high-order four bits of the byte are called a *zone*, except for the rightmost byte in which the sign occupies the zone place.

Referring to the Table of the DKOI codes (see Table 1.1), one can see that the decimal digits are coded as F0 through F9, i.e. as 11110000 through 11111001 in the binary. Therefore, after the input of data from an external device, the zone will be indicated by code 1111. No problem arises when positive decimal numbers are entered in the main storage, as a number, say 125, punched into a card, will be represented after the input operation in the following form

1111 0001 11110010 11110101

Since, as it has been mentioned above, the code 1111 stands for the 'plus' sign, we have in the storage the binary coded decimal representation of a positive binary number +125. Should it be necessary to input the same number, but with the 'minus' sign, we had to replace the last numeral 5 with such a character selected from the DKOI Table which has code 1011 or 1101 in the given part, and code 0101 in the numeric part. Such a character is N (D5). After the data have been prepared on a punched card in the form 12N, their input to the main storage gives

11110001 11110010 11010101  
sign

Arithmetic operations on decimal numbers, however, cannot be carried out on zoned data. The format of the numbers dealt with by the decimal arithmetic instructions is called *packed decimal* and is represented in the form

Digit	Digit	Digit	Digit		Digit	Digit	Digit	Sign
Zero byte		1st byte			14th byte		15th byte	

In the *packed decimal form* two decimal digits are per each byte, except for the rightmost byte of the field. In this byte the sign is to the right of the decimal digit. Both the digits and sign are repre-

sented in the form of four-bit codes. To change decimal data from one format to another, the ES EVM provides appropriate instructions.

It should be noted that, after the execution of any operation of decimal arithmetics, the operation result sign will be always represented by code 1100 for 'plus' and code 1101 for 'minus', regardless of what codes represented the signs of the operands.

Bit strings are certain binary codes which may be from one to 256 bytes long, depending upon the operation being carried out. The information contained in the main storage is interpreted as a bit string in the execution of logic operations and transfer operations.

For example, the same number 0142820D contained in the main storage will be treated:

(a) by an addition command for fixed-point numbers as a positive binary integer equivalent to decimal number 84 542 477;

(b) by an addition command for decimal numbers as a negative decimal number 142 820;

(c) by a logical addition command as a certain binary code.

### **2.3. Classification of Storage Devices**

As has been mentioned in the previous chapter, in the course of computations a necessity often arises of storing certain data with a view to their future use. These functions in a computer are performed by various storage units which widely differ in their parameters, operating principles and in medium. However, all styles of storage units feature three independent modes of operation: they can receive (write mode), retain (short-term or permanent storage), and output (read mode) bit patterns representing data. Prior to considering actual types of storage units, we introduce certain concepts and characteristics by which storage units based on different operating principles can be compared, and classify them.

The storage units may be divided into central (internal) and external stores. The internal storage units include those which are directly dealt with by the CPU. In the ES-1030 computer such storage units are a long-term storage, local storage, protection key blocks, multiplexor channel store, and main storage.

Magnetic drum, magnetic tape and magnetic disk units are referred to as external storage devices.

In this chapter we shall acquaint ourselves in detail with the construction versions and operating principles of main storage, and consider the construction of certain other internal storages. Separate sections of this book will be devoted to the magnetic tape and disk drives.

The storage devices can be also classified by the methods of data writing and reading.

There are two methods of writing: an automatic method without changes in the construction of the drives and non-automatic with changes to the construction of the drives. The first method allows multiple alterations in the data being stored. In the second method, data may be written into the storage only once, during its manufacturing process, and can be altered only by rearrangement of the weaving wires in the storage matrix, wire resoldering, etc. An example of this method of data writing is a long-term storage operating in a computer in the read-only mode.

The main storage permits only automatic writing. If that is the case, the data previously written into a corresponding location (cell) are deleted during a new record in the same location.

Reading data may, or may not destroy the information in storage. Hence, depending upon the style of data readout, the storage devices fall into destructive (full or partial) storages and nondestructive storages. The former type includes, for instance, magnetic core storages. As the data being read may be needed in future, they must be recovered after the read operation. The process of data recovery in a destructive read storage is called regeneration. It should be noted here, that the magnetic core storage also requires that in the write mode the stored data be first cancelled.

The main storages having nondestructive readout may be singled out specially. No regeneration is used in these main memories, as bits do not have to be reversed to be read. Data in a storage location (rectangle) is not erased after any number of readouts, until a command arrives to write new data. Since thin film memories do not have to be returned to their original state after reading, they are faster than magnetic core storages and save some power. The storages with nondestructive readout can be built by depositing very thin rectangles of a nickel ferrite alloy on a glass, ceramic, or metal surface. Unfortunately, it is a very difficult process to build a thin magnetic film fully uniform in structure and magnetic properties. Therefore, if even a main storage is built through the use of thin magnetic films, then to provide reliable data storage, use is still made of the regeneration mode. Nevertheless, the speed of operation of the thin film memories somewhat exceeds that of magnetic core memories with a rectangular hysteresis loop.

As to the data retrieval, the storages may be divided into random access and sequential access storage units.

The *random access storage* is a unit in which equal time is taken by data retrieval from any location (address). This storage provides an automatic access to any location of the storage array. Each location is assigned an address uniquely defining its position in the storage.

In order to retrieve a number from a random-access memory (RAM), the address code is converted into a control pulse which goes directly to the memory locations being retrieved. If the RAM is built

so that a single control pulse is enough to address the retrieved data, then this storage unit is called a Direct Access Storage Device (DASD) in contrast to a computer storage with matching control pulses, in which the address code is converted into two and more control signals. The address buses these signals are applied to are connected to many storage locations. Accessed are, however, solely those of the locations in which the control signals (pulses) are matched. Evidently, this organization of storage imposes additional requirements on the storage locations, i.e. the data in these locations should not be affected (destroyed) by an incomplete set of access pulses applied to them.

One of the advantages of a computer storage with matching of control signals is an essential reduction of the number of address buses. In a DASD the number of output address buses equals the number of words in the storage unit, while the number of address buses in computer storage with matching of two control signals decreases to the square root of the number of words in the storage. A signal on the address bus is generally formed by a powerful amplifier for which reason a decrease in the number of address buses allows the physical size of the storage unit and its address handling equipment to be materially reduced.

*In the sequential access storage device* the address buses are not hard-wired to the storage device, and all locations in time sequence go past the write/read elements. At any given time a special counter points to the location the data may be written in, or read from. The address at which data can be retrieved or written is held in the address register. The address register and current address register are connected by code-comparison circuits, and when the address code matches the status of the current address register, a signal is produced to allow a write or read operation at the address retrieved.

## **2.4. Characteristics of Storage Devices**

Regardless of a very wide diversity of storage types and methods of storage construction, a number of criteria should be singled out in order to be in a position to evaluate any storage unit and to make a comparative analysis of storage units with a view to choosing the type that is best to satisfy the requirements imposed on the computer being designed. These criteria are the characteristics of a storage device that are considered below.

**Storage device capacity.** This is the number of bits which can be stored in the storage device at the same time. Usually, a computer deals with a computer word, and therefore the storage capacity is often specified by the number of words that can be held in its memory array, but in that case the word bit significance must be given. Depending upon the field of application of the storage device, its

design, and the basic storage element, the storage device capacity may vary from several dozens to millions of computer words.

**Storage device access time.** This is the interval between the time the data is called for from a storage device and the time it is made available for processing. This parameter is characteristic of storage operation speed.

Depending upon the type of storage unit, the access time may consist of several constituents. For example, for a magnetic tape storage this time will include the following constituents:

$$T_{ac} = T_{accel} + T_s + T_{r/w}$$

where  $T_{ac}$  is the access time;  $T_{accel}$  is that part of access time in which the tape speed is increased to the speed required for reading or writing;  $T_s$  is the time taken to perform a 'search'; and  $T_{r/w}$  is the time directly taken by data reading or writing.

With the magnetic disk or drum units, the access time consists of two constituents, since the medium is always in motion with respect to the write and read heads:

$$T_{ac} = T_s + T_{r/w}$$

With the internal storage devices in which the address code is directly converted into control pulses, the search time essentially equals the functioning time of the electronic circuits, and the access time of such a storage device may be expressed as follows

$$T_{ac} = T_{r/w}$$

**Data storage time** of a storage device is also one of its basic characteristics. It is an interval during which data can be retained in the storage device without need of refreshing.

This characteristic is closely associated with the reliability of the storage device by which is meant the storage device ability to provide trouble-free operation under certain prescribed conditions.

Besides, like any other devices, the storage units may be evaluated in terms of their dimensions, weight, economy features, etc.

## 2.5. Magnetic Disk Drive. General

The magnetic disk drive (also disk unit, disk handler, disk transport, disk transport unit) is a direct access peripheral storage device that uses magnetic disks as the storage medium. This means that the time of access to any part of data is almost independent of data location.

The cost of a large capacity magnetic core memory is very high so it is advisable to have an auxiliary storage device of a far less cost, though not so fast and convenient as the magnetic core memory. Besides, this auxiliary memory should permit long-term storage of

whatever large arrays of data. The function of such an auxiliary storage can be performed by magnetic tape. The tape, however, has certain disadvantages. An access to data found at the far end of the tape takes much time (dozens of seconds) to seek the data. The data seek on the magnetic disk drives does not exceed 100 to 150 ms. In that case, the maximum amounts of data that can be stored on a magnetic tape reel and a magnetic disk pack are quite comparable. More than that, recently magnetic disk drives are available that employ disk packs of capacities greater than that of magnetic tape reels, with the access time being practically unchanged.

Both, the magnetic disk and magnetic tape drives allow the amount of data stored to be enlarged without limit by changing the magnetic media (tape reels and disk packs). The cost, however, of a magnetic disk pack is essentially higher than that of a magnetic tape reel. Therefore, magnetic tape drives are preferred when large amounts of data are to be stored and used at considerable intervals. It is good practice to keep data of frequent and repeated applications on direct-access storage devices, i.e. on magnetic disk drives.

## **2.6. Applications and Technical Data of Magnetic Tape Units**

The magnetic tape unit (drive) is a peripheral storage device incorporated into a computer model, say, into the ES EVM. The magnetic disk drive is a peripheral storage device that is capable of receiving, retaining and outputting large bulks of data.

We now consider the operating principle of the magnetic tape drive by an example of the drive, type ES-5012M, now furnished as part of the ES EVM system. The other modifications of magnetic tape drives employed by the ES EVM differ, but little, in the characteristics, the theory and principle of operation being the same.

The magnetic tape drive (also tape drive, magnetic tape transport, magnetic tape deck) is a serial access device. The data medium is magnetic tape closely resembling that used on tape recorders. It is typically half an inch wide, with a base of plastic material called mylar and a magnetic coating of iron oxide.

As with a conventional tape recorder in which sound information is converted into an electrical signal which in turn sets up a corresponding pattern on the iron-oxide coating of the tape, the digital data in the magnetic tape drive are converted into electrical pulses to set up minute magnetic spots recorded on the tape surface which represent the bits for coding data. In the latter case, only two states are possible one of which can be accepted by the drive as '0', and the other, as '1'.

The ES-5012M drive employs the non-return-to-zero inverted method of recording.

For the arrangement of data channels on a magnetic tape see Fig. 2.2. Referring to the figure, the information is written byte-by-byte in blocks. Eight bits of each byte plus its parity bit are written in bit-parallel across the magnetic tape. Such cross strings form the record on the tape. The recording density is 8 or 32 characters per 1 mm. Similar bit positions of all bytes (strings) form a channel (nine channels in this case). A reflective strip, called the load point marker, is attached several feet from the end of the tape. At the

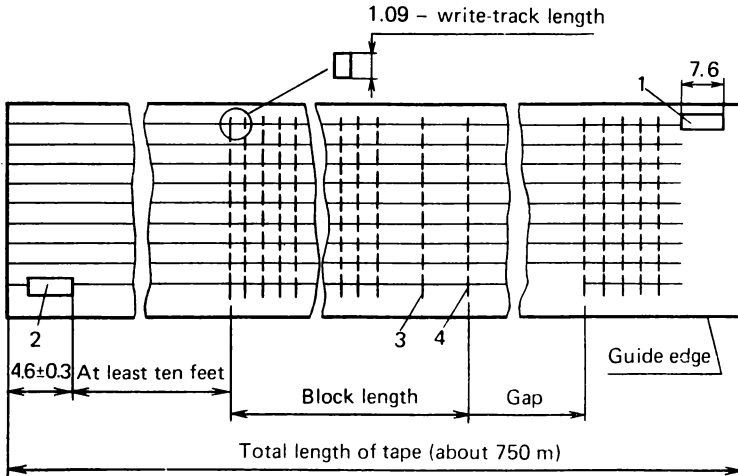


Fig. 2.2. Arrangement of data channels on magnetic tape  
(1) end-of-reel marker; (2) load point marker; (3) cyclic redundancy check (CRC); (4) longitudinal redundancy check (LRC)

opposite end of the tape, an end-of-reel marker is found. The length-wise density of data recording is 32 bits per 1 mm. The capacity of the magnetic tape drive is  $2 \times 10^8$  bits.

Each time a record is written on or read from magnetic tape, the tape must be started and stopped. To allow an instant of time for the tape to begin or stop moving without skipping some data, a gap of blank tape is left between records. This space, called the inter-record (or interblock) gap, is typically .6 or .75 inch wide.

The tape passes the read-write heads at a speed of  $2 \text{ m/s} \pm 5\%$ . A reel of tape (750 m long) takes about 150 s to run through completely at full speed. On the command START or STOP the magnetic tape comes to its stable state at least within 4.5 ms.

The magnetic tape drive is disadvantageous in that it is a serial access device, i.e. records can be written on magnetic tape only in sequential order. Similarly, they can be read back into the computer only sequentially, unlike the case is, say, with the magnetic tape

drives which are direct-access storage devices. The tape rewinding operation is mechanical, for which reason seeking a distant block of data takes seconds, and sometimes dozens of seconds. Therefore, the magnetic tape drive has a lower speed of operation than, for instance, the magnetic disk drive. However, the larger storage capacity of the magnetic tape drive makes it an important component of the external storage equipment of the ES EVM.

## 2.7. Data Organization in the Magnetic Tape Unit

Any spot of magnetic tape to which a bit of data can be written may have the direction of magnetization of its ferric oxide particles changed or not changed, i.e. a 1 or a 0 can be written in it. The nine

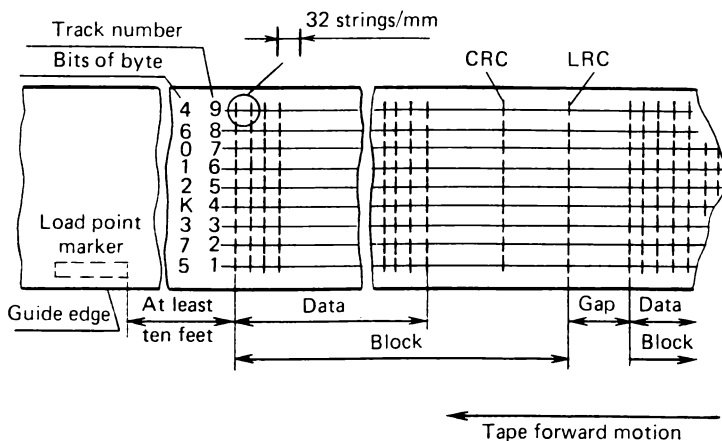


Fig. 2.3. Magnetic tape data organization with record density of 32 bytes per 1 mm (shown oxide side up)

channels of magnetic tape accommodate the DKOI (EBCDIC) code eight bits plus a parity bit, which form a string (byte).

The bytes (strings) are written across the tape. A change in the perpendicular position of bits in a byte is called a skew.

Each byte represents a coded number, letter, or character.

In reading a byte from the tape, all bits must be received simultaneously. A time interval between the instants the first and last bits of a byte are received is called a read skew which can be caused either by a write skew, or time mismatch of the channels in the course of reading.

To make more efficient use of the tape, bytes on the tape are grouped into *blocks* (zones). The minimum length of a block is 18 bytes. The recommended maximum length of it is 2048 bytes.

The magnetic tape drive can record data at two degrees of density: eight bytes per 1 mm and 32 bytes per 1 mm. At the density of 32 bytes (strings) per 1 mm, two check strings are written at the end of each block (Fig. 2.3): a string of cyclic redundancy check (CRC) and a string of longitudinal redundancy check (LRC). The CRC string is formed in the control unit during a write operation and represents the accumulation of all bits in the block. This string is written after the last data byte, over an interval of four bytes. The LRC string

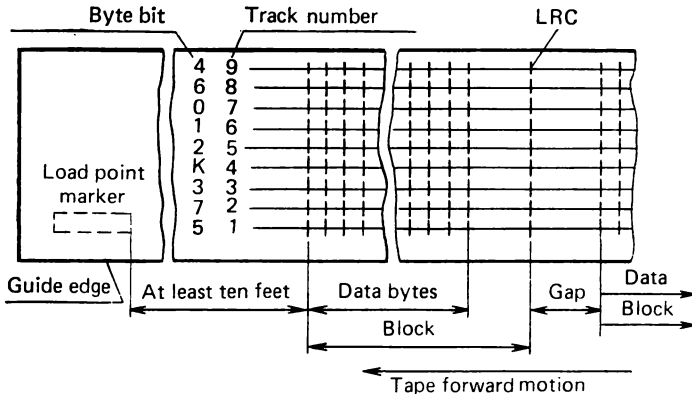


Fig. 2.4. Magnetic tape data organization with record density of 8 bytes per 1 mm (shown oxide side up)

is a count of all bits in each channel of the block. In this case, the total number of bits must be even. This is secured by writing a 1 or a 0 in the LRC string, at the corresponding channel. The LRC string is written on the tape four bytes after the CRC string.

When the record density is eight bytes per 1 mm, only an LRC string is written at the end of each block (no CRC string is written), over an interval of four bytes after the last data byte (Fig. 2.4). As mentioned above, gaps are left between the blocks on tape to allow for the time required to start and stop between reading and writing operations, i.e. gaps defined by hardware factors rather than by the programmer.

Groups of blocks are separated by a block group marker which is one byte block with an LRC string. The data byte and LRC string contain 1's only in bit places 3, 6 and 7.

## 2.8. Fundamentals of the ES EVM Input/Output Devices

All the data input/output devices in the ES EVM operate under control of their control units, which can be implemented in the form of separate devices which is the case with external storage devices, or

as one-piece unit together with the external device under its control. An example is an alphanumerical printer. If that is the case, the control device is often called the control unit of an external device. It should be noted, however, that in both cases the external device is coupled with a channel through a certain control equipment.

The control device in turn is coupled with the channel through the I/O interface. Therefore, the data path from the external devices to the channel is as follows: the external devices—control devices—I/O interface—the I/O channel. The same path is used by the data flow from the channel to the external devices.

Most of the I/O devices incorporated in the ES EVM computers (except, generally, for the external storage devices) have individual

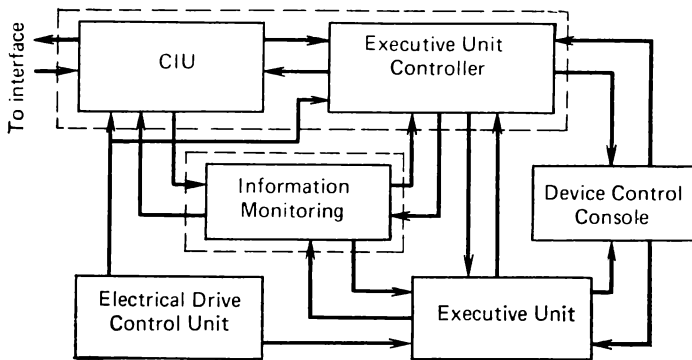


Fig. 2.5. Block diagram of ES EVM I/O devices

control devices, i.e. a control device controls only one external device. In such cases as these, the external device and control device are generally made one-piece. An example is the device of operator-to-computer communication made in the form of a typewriter mounted on a table which also carries a device to control the typewriter.

Most of the I/O devices are electromechanical. All of them include mechanical units which perform mechanical actions on the organization of data input (examples are a mechanism feeding punched cards into a card reader, a typewriter which prints information).

The mechanical unit of the I/O device is controlled by an electronic control unit.

Each input/output device implements a certain method of data input to and output from a computer (by means of punched cards, paper tape, printer, etc.). In compliance with this, each I/O device has its own set of instructions defining the required set of actions to be performed by a given I/O device.

The block diagram of an I/O device is shown in Fig. 2.5. Though the I/O devices may use different operating principles, their confi-

guration is the same. Any I/O device includes an executive mechanism which usually includes a mechanical device (an example is a device feeding punched cards to the card reader). This mechanism performs actual physical actions (feeds punched cards, prints a text, etc.) specified by the control unit for it (for external devices having an individual control unit, the controller is a separate device).

The operation of the I/O device as a whole is under control of the controller of the actuating mechanism. Generally, the controller incorporates a small buffer storage, and sometimes a long-term storage.

In addition to the control of the executive mechanism, the controller of an actual input/output device may perform other functions.

The controller of the executive mechanism is connected to the channel (to receive control information and data) through an interface. This is done however through a channel interfacing unit, rather than directly. The channel interfacing unit (CIU) is a must component of all devices controlling peripherals. It is the last unit next to the interface. Because of the importance of its functions, the CIU is shown in the diagram enclosed in a separate rectangle, though in this diagram it is part of the controller of the executive unit. The functions of the CIU are as follows:

- It receives control pulses from the channel and produces the response pulses;
- It receives device-to-channel interfacing signals from the channel and sends such signals to the channel;
- It decodes the address coming over the lines of the CH-BUS and sends its own address on the A-BUS of the interface to the channel;
- It receives information from and transmits it to the channel;
- It transmits information further to the peripherals, or receives information from them to transmit it to the channel;
- It performs the parity check on the received data;
- It detects certain errors;
- It performs other functions which may not be performed by peripheral devices of all types.

Actual I/O devices in service do not have such a unit as the data monitoring unit. However, many elements performing the data monitoring functions are included in all pieces of equipment comprising peripheral devices and their control devices, i.e. this unit is dispersed all over the device equipment. The control console of the device allows the attending personnel to intervene in the operation of the device, to set the required mode of operation (the off-line operation included), to organize debugging procedures in the off-line operation during adjustment and repair of the device, etc. The power supply control unit controls the power supply to all units of the device.

An I/O device may include many other units which are not compulsory for all I/O devices, or which are parts of the units shown in Fig. 2.5.

## **2.9. Operator's Console**

The operator's console, the principal means of man-computer dialogue, is a component unit inherent in any ES EVM model, regardless of the system configuration. The console allows the operator to supervise the stages of problem solution and state of the system components, to have access to the information stored in the system, and also to enter control information into the computer and receive responses, i.e. to carry on 'conversations'.

The basic equipment of the modern console includes a keyboard for entering messages and responses and a display and/or printer on which messages are received. An operator's console furnished with a display provides reliable information interchange at high speeds in the form suitable for the man. Alphanumeric display units are limited to numbers, letters and special characters. Data may be displayed upon the screen much faster than it can be typed. Some types of display units permit graphic display, so that graphs, drawings, and plotted curves can be shown. The visual display unit (VDU) (also CRT terminal) provides noiseless operation, permits editing and changes in the data format. However, to obtain a copy of some information use should be made of a printer.

Many computers of the previous generations widely employed a console typewriter as the means of communication between the operator and machine. The console typewriters provide easy operation, are relatively small in size and inexpensive. They find their applications in operator's consoles and data preparation devices.

Console typewriters commonly maintain a log of job names, error messages, and other actions taken, which serves as a valuable record of daily operations and source of reference for debugging or troubleshooting.

With small amounts of data transmitted, the low speed of a typewriter is not a disadvantage, since the operator needs time to evaluate the information received and take a decision. This sometimes takes much more time than the typing process.

The principal device of this type most often used nowadays as a component of the ES EVM models is the operator's console based on the typewriter 'Consul 260.1' (Device ES-7077).

## **2.10. Concept of Input/Output Interface**

Of special importance in the study of the ES EVM principles is the study of the input/output interface, as knowledge of fundamentals of the input/output interface operation makes it possible to

understand the whole of the ES EVM data I/O system. Thus, the knowledge of the I/O interface operation allows better insight into the entire computation process occurring in the ES EVM equipment.

Well, what is the input/output interface and what for is it used? To answer the question, recollect how the central part of a computer (the processor, main storage, channels) communicates with the peripheral data input/output devices. In all computers of the Unified System, the data from the main storage to peripheral devices is transmitted through special-purpose devices known as the input/output channels. The channels accomplish input/output operations on commands from the CPU. Meanwhile, the CPU is free from data transfers from the main storage to peripheral devices and back, from peripheral devices to the main storage. It only directs the channels to perform I/O operations and then proceeds directly to the computation work, while the input/output operations are carried out by the channels.

Let us define what is meant by the name peripheral devices. These include:

- All input/output devices, such as a punched card reader, a console typewriter by means of which the operator enters control data into the computer and receives data from it, i.e. a dialogue is carried on between the operator and machine, a paper tape punch/reader, an alphanumerical printer, etc.;

- External storage devices, such as magnetic disk drives, magnetic drum drives, magnetic tape drives, and the like;

- Devices for data communication over telephone and telegraph lines;

- Other feasible facilities of communication between the user and computer.

As you see, there are many diverse styles of peripheral (also external) devices. More than that, progress in technology and computer capabilities require new types of devices for communication between the man and machine. Examples are video-display units (CRT terminals), audio response terminals, voice recognition units, and the like. Naturally, all these external devices differ in operating principles and technique of data transmission to and from the computer, i.e. have different type control pulses at their inputs and outputs. A problem arises, therefore, to enable a device that is in communication with peripheral devices (input/output channels) to exchange information with all the other so much different devices. A solution may be as follows. A special-purpose device known as a device to control a peripheral is engineered for the peripheral devices of each type which could control its peripheral (i.e. to be in a position to control this very peripheral), on the one hand, and could exchange data with the channels, on the other hand.

In this case, devices can be created to control diverse peripheral

devices so that all the different devices controlling peripheral devices will exchange data with the channels in a similar manner suitable for the channels, i.e. the communication line from the control device to the channels and from the channels to the control device may be similar for all the control devices. Meanwhile, the communication line of a control device to the peripheral devices under its control is type-oriented for each physical type of peripherals, i.e. the control devices are matching devices, as it were, and convert the data exchange method of each peripheral device into a method which suits the input/output channels (i.e. the central part of the computer).

Therefore, we may conclude that in case a computer has diverse peripheral devices and incorporates control devices of these peripherals, the data exchange can be so organized that the communication lines from the control devices to the channels (and thus, from the channels to the control devices) may be identical for all the different control devices. It is the input/output interface that tackles this problem, a problem of creating a method of data exchange between the input/output channels and devices used to control peripheral devices of diverse types. By this method is meant the following.

A unified system of communication lines, i.e. use of lines similar in construction and electrical features for communication between the channels and control devices of peripherals, including an equal number of conductors in each line.

A unified system of signals exchanged between the control device and channels, i.e. the same set of data exchange signals and similar electrical parameters of these signals.

Therefore, the interface is so unified, that can be standardized. According to the GOST (USSR State Standard) the full name of the interface in question is the Standard Input/Output Interface of the ES EVM, hereinafter referred to as the I/O interface.

We now lay down the definition of the standard I/O interface.

**Definition.** The standard input/output interface is a unified system of communication links and signals between the input/output channels and control devices (controllers) of peripheral devices.

There are controllers which control several peripherals of similar characteristics at the same time. The latter peripherals are commonly external storage devices. An example is a controller of magnetic disk drives which can handle eight drives concurrently.

Since all the magnetic disk drives are similar and employ the same-type system of their links to the controllers and system of signals, an interface may be also used between the peripheral devices and their control system (in the cases when several peripheral devices are under control of one controller). In contrast to the standard I/O interface between the channels and controller of peripherals, the I/O interface between the controller and peripheral devices is called a small (or internal) interface.

**Definition.** The small I/O interface is a unified system of links and signals found between the controllers of peripheral devices and their corresponding peripherals.

### 2.11. Interface. General

The ES EVM standard I/O interface is compatible with the channels of all ES EVM models. The channels of the unified system computers are designed so that their output parameters correspond to the standard interface.

All devices connected to the channel with the aid of the interface are commonly called *parties*.

The input/output interface provides byte-oriented data transfer from the channels to parties and from parties to the channels. This means that eight data bits are transmitted over the interface lines simultaneously.

The byte being transmitted may be:

- The address of a device the channel wishes to communicate with, and also the communication link from the peripheral device to the channel (several links, if possible);

- A command (in the ES EVM just one byte is assigned to a command code). The command byte is transferred to the party to specify an actual action it must carry out;

- Information about the status of the peripheral device. This information is transferred to the channel from the party upon completion of a certain command stage, or upon completion of the command as a whole. If necessary, the channel is informed about the finally defined status to indicate in more detail the conditions that have caused abnormal situation, or a failure in the peripheral device or in the controller of the peripheral device;

- A specific slice of data proper (if at that instant data exchange is taking place between the channel and party, rather than control information transfer). Data in this event stand for the information read from the peripheral device for transfer to the main storage or read out of the main storage to the peripheral device. An example is information writing from the main storage onto magnetic disks.

## CHAPTER 3

## PROGRAMMING. GENERAL

### 3.1. Algorithm Concept

The algorithm concept is one of the principal notions of mathematics and computer engineering. The word *algorithm* may be defined as a set of unambiguous rules that define how a particular problem,

or class of problems, can be solved in a finite sequence of steps.

The most simple algorithms people encounter during their studying are the rules for executing arithmetic operations. The word algorithm is derived from the name of an Arabian mathematician, al-Khowarizmi who lived in the ninth century and formulated those rules.

Algorithms are generally regarded as being either nonnumerical or numerical. Nonnumerical algorithms are those that do not involve calculations. Numerical algorithms involve some type of mathematical calculation. Let us consider the execution of an addition operation as an example of a numerical algorithm.

The process of addition is broken down into a sequence of simple operations during the execution of which the calculator deals only with two similar summand digits, the order of performing these elementary operations being specified forever (from right to left). The elementary operations of addition are subdivided into two types. These are writing down the corresponding digit of sum according to the table of digit addition, and showing a unit carry-over to the adjacent more significant digit position made above the left neighbouring digit.

The algorithms of the other three arithmetic operations are fairly simple too.

The algorithm of finding all prime numbers less than prescribed number  $N$  may be considered as another example. A method for finding the prime numbers less than any given number was suggested by a Greek mathematician named Eratosthenes (276-195? B.C.). This method consists of sifting out those numbers which are not prime numbers and is called the sieve of Eratosthenes.

Let us construct the algorithm to find all prime numbers less than 30. First, write all the integer numbers from 1 to 29:

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	

and follow the instructions below.

*Instruction 1.* Cross out the number 1, since it is not a prime number. Proceed to instruction 2.

*Instruction 2.* Cross out all the numbers that have 2 as a factor, except 2 itself. This gives us:

<del>1</del>	2	3	<del>4</del>	5	<del>6</del>
7	<del>8</del>	9	<del>10</del>	11	<del>12</del>
13	<del>14</del>	15	<del>16</del>	17	<del>18</del>
19	<del>20</del>	21	<del>22</del>	23	<del>24</del>
25	<del>26</del>	27	<del>28</del>	29	

and proceed to instruction 3.

*Instruction 3.* Cross out all the remaining numbers that have 3 as a factor, except 3 itself. This gives us:

<del>1</del>	2	3	<del>4</del>	5	<del>6</del>
7	<del>8</del>	<del>9</del>	<del>10</del>	11	<del>12</del>
13	<del>14</del>	<del>15</del>	<del>16</del>	17	<del>18</del>
19	<del>20</del>	<del>21</del>	<del>22</del>	23	<del>24</del>
25	<del>26</del>	<del>27</del>	<del>28</del>	29	

and proceed to instruction 4.

*Instruction 4.* The next number which is not crossed out is 5; now cross out all the remaining numbers that have 5 as a factor, except 5 itself. This will give us:

<del>1</del>	2	3	<del>4</del>	5	<del>6</del>
7	<del>8</del>	<del>9</del>	<del>10</del>	11	<del>12</del>
13	<del>14</del>	<del>15</del>	<del>16</del>	17	<del>18</del>
19	<del>20</del>	<del>21</del>	<del>22</del>	23	<del>24</del>
<del>25</del>	<del>26</del>	<del>27</del>	<del>28</del>	29	

Those numbers which are not crossed out are the prime numbers less than 30. The set is 2, 3, 5, 7, 11, 13, 17, 19, 23, 29. Note that the finding process terminates after crossing out all the numbers that have a maximum integer not greater than  $\sqrt{N}$  (5 in this case) as a factor.

One more example is the 'Euclid Algorithm' used to solve the following problem: find the greatest common divisor (GCD) of two prescribed natural numbers  $a$  and  $b$ . The algorithm of finding the greatest common divisor is as follows.

*Instruction 1.* Compare the numbers  $a$  and  $b$ . Two cases can result:  $a = b$ , or  $a \neq b$ . If  $a = b$ , take any one of them as the GCD. Stop the calculation. If  $a \neq b$ , proceed to Instruction 2.

*Instruction 2.* If  $a < b$ , proceed to Instruction 3, if  $a > b$ , proceed to Instruction 4.

*Instruction 3.* Exchange  $a$  and  $b$ . Proceed to Instruction 4.

*Instruction 4.* Subtract the smaller number from the greater number. Proceed to Instruction 5.

*Instruction 5.* If the difference equals 0, take the subtrahend as the GCD. Stop the calculations. If the difference is other than 0, proceed to Instruction 6.

*Instruction 6.* Take the difference as the new subtrahend, and the old subtrahend as the new minuend. Proceed to Instruction 1.

Referring to the definition of an algorithm and examples considered, we can single out the following features inherent in any algorithm.

**Determinancy of an algorithm.** Since the nomenclature of rules comprising an algorithm is fully unambiguous, its repeated application under the same start conditions yields the same result.

**Wide use of an algorithm.** An algorithm is not a prescription to solve one particular problem, but allows solution of a whole class of problems of the same type under different starting conditions.

**Specific result of an algorithm.** Being a set of rules or procedural steps that are to be followed in sequence, an algorithm either solves a particular problem or produces a signal that the problem cannot be solved.

**Problems.**

1. Construct an algorithm to solve a linear equation in one unknown  $y = ax + b$ .
2. Construct an algorithm to multiply two numbers represented in the binary notation.

### 3.2. Specification of Algorithm

This section deals only with some techniques of algorithm specification.

Considering the definition of an algorithm in the previous section, we resorted to a word description of an algorithm. This representation of algorithms is unsuitable for entry into a computer. This should be done in a machine language so that the computer is automatically told what to do in the course of a given problem solution. An algorithm written in a machine language is known as a *program of problem solution*.

However, to write algorithms for solution of complicated problems at once in the form of a program in the machine language is very inconvenient, as in this event the problem is difficult to grasp which hinders with the understanding of the structure of a complicated algorithm and separating of essential links from inessential. Errors are most likely to occur under such circumstances which are difficult to detect and correct in this style of program writing. Generally an algorithm is constructed in several attempts with returns back to correct errors and make the structure of the algorithm more precise. A convenient way to represent algorithms is provided by a graphical diagram commonly used to assist the programmer in the first stages of the algorithm structure. This diagram is called a flowchart or flow diagram. A *flowchart* consists of a number of blocks, each being a geometrical figure (a rectangle, circle, polygon, etc.) and representing a single conceptual step in the process. These are connected by lines to show the sequence in which they are to be performed: the flow lines may be arrowed to show the direction of flow; but the standard convention is to assume that the direction of flow is from top to bottom and left to right, and to put arrows only on

the lines oriented in the opposite sense. Sometimes, the arrows are supplemented with an inscription to indicate when this direction is chosen.

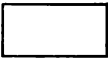
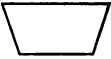
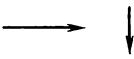

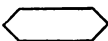
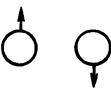

The process action is indicated in the block symbol by a statement; a large statement is given in a legend to the flowchart with the statement number placed inside the symbol. The statement may be written in plain language, or expressed algebraically, or given in some special symbology. Table 3.1 shows the accepted set of symbols used in flowcharts.

Most generally, we may single out the following groups of blocks.

1. The process block. This is associated with all calculations, data movement, initialization, and program-modification steps.

2. The input/output block. This includes a group of instructions to perform the functions of readdressing, recovery, data transfer to

**Table 3.1**

Block symbol	Functions
	PROCESS. Any calculation, data movement or other processing function in a program
	INPUT/OUTPUT. Any read, write, or control function of an input/output device
	SEQUENCE OF OPERATIONS. Indication of logic flow
	DECISION. A test of some condition to determine which of two or more alternate paths should be taken
	PREDEFINED PROCESS. A group of instructions such as a subroutine, spelled out in detail elsewhere
	CONNECTOR. An entry to, or exit from, another part of the program flowchart
	TERMINAL. The beginning, end, or point of interruption of program

special locations, clearing working locations of storage, etc. In other words, this part of the program performs the functions of preparing the other blocks for the beginning of the computation process. Usually, these blocks are used for those processes of computation in which the computations are made according to the same formulas, but for different source data.

3. The decision block. The diamond-shaped decision block is used to express any condition that might provide alternate paths of flow

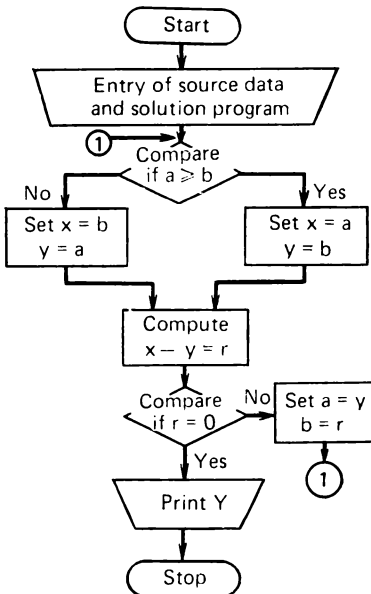


Fig. 3.1. Flowchart of algorithm for finding GCD

through the program. If the condition is based on a yes or no possibility, two exit lines show the possible path that the program may take. If the condition is based on a comparison between the relative size of two numbers or values, three exit paths are provided, depending upon whether the first number is less than, equal to, or greater than the second number.

4. The predefined process. This is widely used in diverse programs when we wish to use a subroutine, or a series of steps for some particular purpose, which is repeated several times throughout the program. It is customary to write the logic for the subroutine in detail at some point in the program and then use the predefined process symbol wherever the subroutine might be used at other points in the pro-

gram. Examples may be groups of instructions performing data input to a digital computer, conversion from decimal to binary, from binary to decimal, dump, machine halt, a routine that calculates a square root, etc.

The above classification is conventional, as, strictly speaking, all these blocks are computational.

Examples of flowcharts with the above-mentioned set of blocks are given below.

Let us consider, as an example of using the designations listed in Table 3.1, the flowchart of an algorithm for finding the greatest common divisor (GCD) of two numbers  $a$  and  $b$  (Fig. 3.1).

In the flowchart shown, the diamond-shaped block is for taking decisions at points where alternative course of action must be taken

according to the value of some item of data at that point. When the condition stated in the diamond-shaped block is met, follow the path indicated by 'yes', if not met, follow the 'no' path. If this stage has always one and the same continuation, it is designated by a rectangle.

The basic merit of the algorithm flowchart is that it represents algorithms in a convenient graphical way. With many details however the flowchart becomes awkward and deceptive.

Note, that the use solely of an algorithm flowchart is not always sufficient to execute the algorithm depicted. We must also use English terms, mathematical notation, or algorithmic language to show the operation performed and the specific data items involved. In this event, the flowchart much assists the programmer in his work.

The *algorithmic language* is a set of symbols representing the language alphabet, a system of rules for coupling symbols into words by means of which separate language constituents are represented (the language syntax), and a system of rules for interpreting the relationship between symbols and their meanings (semantics).

Let us consider four language elements, namely the symbols, words, expressions, and statements.

*Symbols* of language are basic indivisible characters used to write texts in the language.

*Words* of language. A word is a character string that has significance as a minimum unit in a language. Thus, a word is a combination of characters followed by a blank, or a punctuation mark. In this sense, a number is also a word. However, the printing devices of computers are designed so that their outputs are lines of letter and digit characters, rather than separate words. In this case, words, i.e. lines (strings) of letters and digits are called *identifiers*, *variables*, *labels*, or *numbers*.

*Expressions* are groups of words or parts of sentences of a language. In the algorithmic languages expressions are assigned a certain value.

The *statement* of an algorithmic language is a brief description of one or more operations that are to be performed during processing.

At present, the tendency of automating the programming process has resulted in the creation of international algorithmic languages such as FORTRAN, Assembler, PL/1, and others.

### 3.3. Computation Process According to Block Diagram of Digital Computer

We now consider the computation process of a problem from the instant the program has been prepared, debugged, translated, and entered into the library. A block diagram of a computer is shown in Fig. 2.1. The procedure of solving a problem on a digital computer is

started with calling the program from the library to the main storage and entering the source data on the problem.

As a rule the library is stored on a magnetic disk in one of the direct-access magnetic disk drives NMD0 through NMD8. The choice of the magnetic drive is performed by the magnetic disk drive controller which is connected through the first channel to the main storage and central processor unit. At the beginning of program run, preparatory operations are commonly carried out to clear the required locations of the main storage and the assigned tracks (channels) on the direct-access devices. Next, the data are entered from one of the card readers or paper tape reader (or magtape drives NML0 through NML8, if available) and processed. The corresponding channel and computer device are selected with the aid of the controllers, as dictated by the data medium. After the entry, the data are converted and processed in the main storage and CPU.

The program is executed in sequence on instructions from the central processing unit.

Upon completion of the computational process, the results of the problem solution are stored on one of the data media (punched cards, paper tapes) or are printed on paper.

During the problem solution, time messages are transmitted to the console (the device incorporating the Consul typewriter) to tell the operator about the run of the computation process, and program and machine errors, if any. Upon completion of the problem solution, the operator is given an appropriate message with indication of the solution end time and time taken by the problem solution.

## PART 2

# Algorithmic Languages

## CHAPTER 4

### INSTRUCTION SET

#### 4.1. Instruction Format

Typical ES EVM instruction consists of one *operation code* (opcode) telling the computer precisely what to do, and two addresses as operands which are the data that will be acted upon. The operand can be located in one of the 16 general-purpose registers (hereinafter referred to as general registers, or registers) in the main storage, or is specified in the instruction itself. In the latter case, it is called an immediate operand.

There are five types of instructions in the instruction set. These types are designated by the following format codes: RR, RX, RS, SI and SS. The format codes indicate the place of operand location: R stands for an operand in a general register; X is an operand in an index register (halfword, word, or double word, depending upon the type of instruction) in the main storage; S is an operand of arbitrary length in the main storage; I is an immediate operand. Depending upon the format, instruction length is one, two, or three halfwords. In the storage an instruction should begin at a halfword boundary.

Two hexadecimal digits (a byte) are required to represent an operation code. An 8-bit code allows up to 256 separate operations to be represented in one storage location, or byte. However, the computer deals with somewhat more than half the number of possible operation (OP) codes.

The address of an operand in a register is specified by one hexadecimal digit, since the computer has only sixteen registers.

The addresses of operands in the main storage are specified not so easily. Since, we have to know how to address any of  $2^{24}$  bytes of the main storage, 24 bits (six hexadecimal digits) will be needed to write an address. However, with this method of writing addresses, the program will contain much of unnecessary information. An average length of a program is several hundreds or thousands of bytes. Therefore, several leftmost digits in all addresses of the main storage which are occupied by the program will be similar. To save the main storage space, it is good practice to write addresses as short as possible. In the ES EVM the operand addresses are written in the

form

X B DDD

—for the operands with the X format code, or in the form

B DDD

—for the operands with the S format code.

X and B are hexadecimal digits equal to 0, or stand for one of the general registers numbered 1 through 15. Register X is called an *index register* and its contents, an *index*. Register B is designated as a *base register*, and its contents, as a *base*. The DDD, the three hexadecimal digits, are called a *displacement*. Hereinafter a displacement will be designated by one letter D.

The address of the main storage is formed by adding the displacement and base. In the case of operands with the X format, the contents of the index register and the base register are added to the displacement. The computed value is called an *effective address* and designated by the letter E.

Symbolically it may be written as follows

$$E = (X) + (B) + D$$

where the parenthesized letters stand for the contents of the respective register. In execution of addition, all numbers are taken as binary integers, and if the sum occupies more than three bytes, its six rightmost digits are taken as the effective (actual) address.

Special attention should be paid to the use of general register 0. If  $X = 0$  or  $B = 0$ , then in forming the effective address, use is made of value 0, rather than the contents of register 0. In this case, they speak of the absence of base or index.

**Example.** This example illustrates formation of effective addresses in obeying the instructions.

Register 0	00 67 4F 3C
Register 1	00 03 A0 00
Register 12	00 00 00 5E
Address in the instruction (B DDD)	1 576
Effective address	03A576
Address in the instruction (X B DDD)	1 C 008
Effective address	03A066
Address in the instruction (X B DDD)	0 0 216
Effective address	000216

Figure 4.1 shows the format of the five different machine instructions. In all the instructions the first byte is occupied by the opcode (OP). In the RR format,  $R_1$  and  $R_2$  are the addresses of the registers containing the first and second operands, respectively. In the RX

format,  $R_1$  is the address of the register containing a first operand, and  $X_2$ ,  $B_2$  and  $D_2$  define the address of the second operand in the main storage. The instructions of formats RR and RX are most popular.

There are several instructions in the ES EVM which require three rather than two operands to be obeyed. To this end, use is made of format RS which looks just like the RX format. The only difference is that the rightmost four bits of the second byte refer to a general register used as a third operand rather than as an index register for the second operand.

In the instructions of format SI the first operand ( $B_1$ ,  $D_1$ ) is in the main storage. The second operand is immediate. It is in the instruction itself. Field  $I_2$  defines *one byte* of data which participates in the operation.

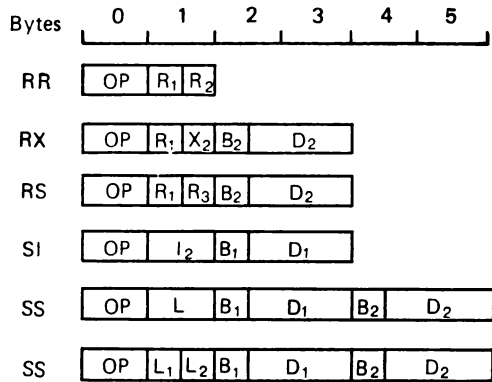


Fig. 4.1. Formats of machine instructions

There are two versions of the SS format instructions. In the first of these versions, L is the length of operands which indicates how many bytes participate in the operation. The operands should have equal lengths within the range from 1 to 256 bytes. The L is always one less than the number of bytes participating in the operation and may take a value from 0 to 255.

The second version of the SS format is intended for instructions whose operands may differ in length. Therefore, each operand should have its own length code ( $L_1$  for the first operand, and  $L_2$  for the second operand), occupying one hexadecimal digit. In this version of the SS format, operands may be from 1 to 16 bytes in length:  $L_1$  and  $L_2$  should be one less than the length of the corresponding field and may take values from 0 to 15.

## 4.2. Arithmetic Fixed-Point Operations

The instructions may be classified both by the format, and in compliance with the data on which an operation is executed. The fixed-point instructions are for actions on fixed-point numbers a word or halfword in length. The operands of fixed-point instructions should be in the main storage beginning at the boundary of word or halfword, depending upon the length of the operand. These instru-

ctions use general register whose capacity of 32 bits is in compliance with the length of fixed-point numbers.

In the instructions we shall indicate:  
 instruction name, for example, ADD  
 mnemonic designation, for example, A  
 instruction format, for example, RX  
 operation code, for example, 5A  
 instruction description.

Since digital codes of instructions are difficult to remember, we shall mainly use their mnemonic designations (codes). These designations are also used when programming in the Assembler language. Let us consider several instructions written in this manner.

LOAD 58 L RX  $R_1X_2B_2D_2$   $(E_2) \rightarrow R_1$

A word is fetched from the main storage at the address of the second operand and stored in the place of the first operand. The previous content of register  $R_1$  remains unchanged. Effective address  $E_2$  should begin at the word boundary, i.e. must be a multiple of four.

LOAD 18 LR RR  $R_1R_2$   $(R_2) \rightarrow R_1$

The instruction is similar to the previous one. The only difference is that the second operand is in the register. Recall, that parentheses are used to designate contents. Thus,  $(E_2)$  stands for the number stored at effective address  $E_2$  and  $(R_2)$  means the contents of register  $R_2$ .

As with the LOAD instruction, many other instructions have several modifications, depending upon how the operands are arranged. In future, we shall give such instructions together with one description.

ADD 5A A RX  $R_1X_2B_2D_2$   
 $(R_1) + (E_2) \rightarrow R_1$   
 ADD 1A AR RR  $R_1R_2$   
 $(R_1) + (R_2) \rightarrow R_1$

The second operand is added to the first operand. The sum is substituted for the previous value of the first operand, and the second operand remains unchanged.

SUBTRACT 5B S RX  $R_1X_2B_2D_2$   
 $(R_1) - (E_2) \rightarrow R_1$   
 SUBTRACT 1B SR RR  $R_1R_2$   
 $(R_1) - (R_2) \rightarrow R_1$

The second operand is subtracted from the first operand. The difference is substituted for the previous value of the first operand. The second operand remains unchanged.

STORE 50 ST RX  $R_1X_2B_2D_2$   
 $(R_1) \rightarrow E_2$

The contents of general register  $R_1$  are written in the main storage at effective address  $E_2$ . The first operand remains unchanged. This instruction is converse of load (L). There is no format RR for it.

**Example.** This example illustrates the change occurring in the contents of the registers and main storage during execution of the above-described instructions.

								Register 1	00	00	07	00
								Register 2	00	00	08	FF
								Register 3	00	00	16	A8
								Register 4	FF	FF	FA	41
								Storage 000720-23	00	00	29	19

In this and in the examples that follow, the right-hand column contains only the changes resulting from the execution of the corresponding instruction. The contents of the other registers and fields of the main storage remain unchanged.

MULTIPLY      5C M RX  $R_1X_2B_2D_2$   
 $(R_1 + 1) \times (E_2) \rightarrow R_1, R_1 + 1$   
 MULTIPLY      1C MR RR  $R_1R_2$   
 $(R_1 + 1) \times (R_2) \rightarrow R_1, R_1 + 1'$

Multiplication differs from addition and subtraction in that it may lack one register to write the product. Thus, if two 31-bit numbers are multiplied together, the product may occupy up to 62 bits, for which reason the product is situated in two adjacent registers: in register  $R_1$  specified in the instruction and in register  $R_1 + 1$  having a number one more. The 32 low-order bits of the product are written in the  $R_1 + 1$  register, and the high-order bits are written in the  $R_1$  register. The product sign is only in the  $R_1$ , since the sign bit of the  $R_1 + 1$  register is part of the product. Such a number occupying two registers (or a double word in the main storage) is defined as a *double precision number*.

Note, that the multiplicand is not in the register  $R_1$  specified in the instruction. It is in the register having a number incremented by 1.

General register  $R_1$  should have an even number. If the register specified in the instruction is odd, this will be handled as an error.

and the instruction will not be obeyed. This is because number  $R_1 + 1$  in the execution of the instruction must be computed by number  $R_1$ . This operation is easy with even  $R_1$ , since to add one to an even number means to substitute 1 for 0 in the LSB. However, to add 1 to an odd number requires a far more intricate algorithm. This limitation is used to simplify the circuit of the arithmetic unit.

### Example

		Register 0	09 AB CD EF
		Register 1	00 00 00 26
		Register 2	FF FF FF DA
		Register 3	02 00 00 05
		Register 4	00 00 60 00
		Storage 0060AC-AF	00 00 00 12
<i>Instruction</i>		<i>After execution of instruction</i>	
M	5C 0 0 4 0AC	Register 0	00 00 00 00
		Register 1	00 00 02 AC
MR	1C 0 2	Register 0	FF FF FF FF
		Register 1	FF FF FD 54
MR	1C 2 3	Register 2	00 04 00 00
		Register 3	14 00 00 19

Recall that the multiplicand is not in the register specified in the instruction, but in a register whose number is 1-incremented. Hence, the last instruction in the example multiplies the number in register 3 by the same number.

DIVIDE	5D D RX
50 $R_1X_2B_2D_2$	$(R_1, R_1 + 1) : (E_2)$
	quotient $\rightarrow R_1 + 1$
	remainder $\rightarrow R_1$
DIVIDE	1D DR RR
1D $R_1R_2$	$(R_1, R_1 + 1) : (R_2)$
	quotient $\rightarrow R_1 + 1$
	remainder $\rightarrow R_1$

Like in the MULTIPLY instruction, the  $R_1$  must be even. The dividend is a double-precision number stored in adjacent registers  $R_1$  and  $R_1 + 1$ . The divisor is of conventional length and occupies a word (or a register).

Both operands are handled as integers. The division is made evenly. The quotient is stored in register  $R_1 + 1$ , and the remainder, in register  $R_1$ . The quotient is signed according to the conventional algebraic rules. If not equal to 0, the remainder has the same sign as the dividend.

The programmer should take care to make one register (31 bits) sufficient for writing the quotient, otherwise an error will be recorded and the DIVIDE operation will not be carried out.

**Example**

		Register 0	FF FF FF FF
		Register 1	FF FF FD 54
		Register 2	00 00 00 00
		Register 3	00 00 02 AE
		Register 4	00 00 00 26
		Register 5	00 00 70 10
		Storage 007014-18	FF FF FF DA
	<i>Instruction</i>	<i>After execution of instruction</i>	
D	5D 0 0 5 004	Register 0	00 00 00 00
		Register 1	00 00 00 26
DR	1D 2 4	Register 2	00 00 00 02
		Register 3	00 00 00 26

**Exercises**

1. Explain why the operands of fixed-point instructions must begin at the full word boundary when stored in the main storage.
2. Write an instruction which will move a zero code to register 5 regardless of its previous contents.
3. General register 1 contains number X. How can you store number  $-X$  in register 1 with the aid of the instructions you know?
4. In the main storage number A is at address 000804 and number B at address 000808. Write a program which will store the value of  $A^8 - B^8$  in general register 2.
5. Suppose (conventionally) that the fixed-point instructions studied have the following execution time:

$$\begin{aligned}
 \text{LR, AR, SR} &= t \\
 \text{L, A, S, ST} &= 2t \\
 \text{MR} &= 4t \\
 \text{M, DR} &= 5t \\
 \text{D} &= 6t
 \end{aligned}$$

where  $t$  is a certain period of time dependent upon the model of computer.

Write a program which, being used under the conditions stated at point 4, will (a) take the smallest space in the storage, and (b) be the fastest.

Will you be able to reach 22 bytes in case (a) and  $27t$  in case (b)?

**4.3. Programming Techniques**

Any program can be written by many methods. In this section, we shall consider the basic principles underlying the quality of a written program and allowing a choice of best programming techniques. These principles are of a general-purpose nature. When applied to any programming section these principles give rise to its specific

techniques; programming skills are based on the thorough mastering of these techniques.

We shall formulate the principles by writing a program for the following calculation

$$Y = X^4 - 4X^3 + 6X^2 - 5X + 1$$

Let  $X$  and  $Y$  be whole numbers and suppose for simplicity that both the final result and all intermediate results are stored in one register.

First allocate the main storage (with the address in the left-hand column and the contents in the right-hand column):

004000-03	X
004004-07	Y
004008-0B	00 00 00 04
00400C-0F	00 00 00 06
004010-13	00 00 00 05
004014-17	00 00 00 01

Let register 15 be a base register and suppose that it contains the value of 4000.

Now, it is necessary to determine the order of computation operations.

*Method 1* does not require special considerations and consists in sequential computation of each summand: first  $X^4$ , then  $-4X^3$ , etc. after which they are summed up. This method is far from being satisfactory, as in raising  $X$  to high-order powers, we concurrently raise it to low-order powers, but the values resulting from the latter operations are lost, and we have to repeat these operations. On this basis, the first principle of programming may be formulated as follows.

**Principle 1.** Never do a job twice. Make attempts to utilize a previous result.

*Method 2* is free from the above demerit and consists in processing the summands in the reverse sequence. In this event, raising  $X$  to high-order powers, say  $X^3$ , we may utilize the previously computed value of the low-order power ( $X^2$ ). Next, process all the five summands and add them together.

**Principle 2.**<sup>4</sup> Do not store the partial (intermediate) results more than necessary. Utilize these results at once, as it is possible.

*Method 3* consists in that, having processed a sequential summand, we add it to the sum of the previous summands, rather than store it until all the other summands have been processed.

Study the program below with due attention.

L	58	1	0	F 014	$1 \rightarrow R_1$	
L	58	3	0	F 000	$X \rightarrow R_3$	
L	58	5	0	F 010	$5 \rightarrow R_5$	
MR	1C	4	3		$(R_5) \times (R_3) \rightarrow R_5$	$(R_5) = 5X$
SR	1B	1	5		$(R_1) - (R_5) \rightarrow R_1$	$(R_1) = 1 - 5X$
M	5C	2	0	F 000	$(R_3) \times X \rightarrow R_3$	$(R_3) = X^2$
L	58	5	0	F 00C	$6 \rightarrow R_5$	
MR	1C	4	3		$(R_5) \times (R_3) \rightarrow R_5$	$(R_5) = 6X^2$
AR	1A	1	5		$(R_1) + (R_5) \rightarrow R_1$	$(R_1) = 1 - 5X + 6X^2$
M	5C	2	0	F 000	$(R_3) \times X \rightarrow R_3$	$(R_3) = X^3$
L	58	5	0	F 008	$4 \rightarrow R_5$	
MR	1C	4	3		$(R_5) \times (R_3) \rightarrow R_5$	$(R_5) = 4X^3$
SR	1B	1	5		$(R_1) - (R_5) \rightarrow R_1$	$(R_1) = 1 - 5X + 6X^2 - 4X^3$
M	5C	2	0	F 000	$(R_3) \times X \rightarrow R_3$	$(R_3) = X^4$
AR	1A	1	3		$(R_1) + (R_3) \rightarrow R_1$	$(R_1) = 1 - 5X + 6X^2 - 4X^3 + X^4$
ST	50	1	0	F 004	$(R_1) \rightarrow Y$	

The sum value is accumulated in register 1, register 3 is for raising  $X$  to the sequential power, and register 5 is used for computation of the sequential summand. Registers 2 and 4 are also used by the program.

**Principle 3.** Do your best to save the program execution time, storage space occupied and other hardware facilities. Of equivalent methods select the fastest one requiring less storage, fewer registers, etc.

To carry out each arithmetic operation, there are two instructions in the computer. These are in format RX (register to indexed storage) and format RR (register to register). The instructions of register RR are faster, since they do not perform the following steps of instruction execution: computation of an effective address and reading of the second operand from the main storage. However, to execute an instruction of format RR, the second operand should be first loaded to the register. Therefore, if one and the same datum is utilized in several instructions, it is better to load it once in the register, and then to use the instructions of format RR.

*Method 4* consists in that  $X$  is once loaded in a register and then use is made of its value from the register. In this case, the program will be faster.

**Principle 4.** Do not do, what is done. Use and accumulate standard techniques of programming.

*Method 5.* Let our formula be as follows

$$Y = (((X - 4) X + 6) X - 5) X + 1$$

Now, we shall carry out the operations in sequence starting with the most internal parentheses. This is a standard method for computation of values of any polynomials. The computation program is called the Horner method.

L	58 1 0 F 000	$X_1 \rightarrow R_1$	
LR	18 2 1	$(R_1) \rightarrow R_2$	$(R_2) = X$
S	5B 1 0 F 008	$(R_1) - 4 \rightarrow R_1$	$(R_1) = X - 4$
MR	1C 0 2	$(R_1) \times (R_2) \rightarrow R_1$	$(R_1) = X^2 - 4X$
A	5A 1 0 F 00C	$(R_1) + 6 \rightarrow R_1$	$(R_1) = X^2 - 4X + 6$
MR	1C 0 2	$(R_1) \times (R_2) \rightarrow R_1$	$(R_1) = X^3 - 4X^2 + 6X$
S	5B 1 0 F 010	$(R_1) - 5 \rightarrow (R_1)$	$(R_1) = X^3 - 4X^2 + 6X - 5$
MR	1C 0 2	$(R_1) \times (R_2) \rightarrow R_1$	$(R_1) = X^4 - 4X^3 + 6X^2 - 5X$
A	5A 1 0 F 014	$(R_1) + 1 \rightarrow R_1$	$(R_1) = X^4 - 4X^3 + 6X^2 - 5X + 1$
ST	50 1 0 F 004	$(R_1) \rightarrow Y$	

The value of  $Y$  is sequentially computed in register 1, and the value of  $X$  is stored in register 2. The program also utilizes register 0.

The program has undergone material evolution. The reader may write a program according to method 1, and having compared it with the latter version, he will convince himself of the necessity to give some time to speculations and analysis of the problem in question, before an attempt to write a simplest program.

**Principle 5.** Use all data of the problem. Often solution of an actual problem is not so complicated as the common solution of a class of problems to which the problem belongs.

The latter program is suitable for computing any polynomial (expanded numeral), it is only necessary to change the coefficients. However, our problem is to compute the value of a polynomial. Is there an easier way than in the common case?

Represent the polynomial as follows  $Y = (X - 1)^4 - X$ .

Now the program becomes entirely simple:

	004000-03	X	
	004004-07	Y	
	004008-0B	00 00 00 01	
	Register 15	00 00 40 00	
L	58 1 0 F 000	$X \rightarrow R_1$	
S	5B 1 0 F 008	$(R_1) - 1 \rightarrow R_1$	$(R_1) = X - 1$
MR	1C 0 1	$(R_1) \times (R_1) \rightarrow R_1$	$(R_1) = (X - 1)^2$
MR	1C 0 1	$(R_1) \times (R_1) \rightarrow R_1$	$(R_1) = (X - 1)^4$
S	5B 1 0 F 000	$(R_1) - X \rightarrow R_1$	$(R_1) = (X - 1)^4 - X$
ST	50 1 0 F 004	$(R_1) \rightarrow Y$	

The program employs registers 0 and 1.

Well, suppose that the values of the polynomial coefficients have changed (which is often the case in practice). This will show us at once all advantages of the standard solution with the help of the Horner method. It is far from advisable to seek artificial methods in all cases.

**Principle 6.** Seek non-standard methods only when standard methods fail to yield satisfactory results. For example, when they are slow, or require too much space of storage, etc.

Now, note the principal aspects which should be paid special attention to in programming computations according to the algebraic formulas with the use of the fixed-point arithmetic operations studied. These state that:

- computations should be carried out in the order requiring a least number of partial result expressions;
- common subexpressions should be found and processed once;
- registers should be used completely, without accessing the main storage several times for the same datum;
- odd registers should be used for data that will be multiplied or divided by other data, with no unnecessary data transfers.

#### 4.4. Branching

No practical computer program can be written without the use of decision blocks, which give the programmer the possibility of specifying alternative courses of action depending on the outcome of some test. Instructions giving this facility are known as JUMP or BRANCH. Though, the instructions in a program are generally executed following the sequence they are written in, when computations are other than according to formulas, we often have to make the computer modify its operations. Thus, branching is used any

time we wish to change the sequence of instructions in a program, depending on some conditions.

Let us write a program

$$X = \begin{cases} A^2 + B^2 - A & \text{at } A > B \\ A^2 + B^2 + A & \text{at } A \leq B \end{cases}$$

We may compute both expressions, but we still have to choose the one of them we need. To do this, the computer has special facilities.

In the execution of certain instructions a *condition code* (CC) is produced. It is an indicator which may assume the values of 0, 1, 2 or 3. The ADD and SUBTRACT (A, AR, S, SR) instructions produce CC = 3, if an *overflow* occurs in the execution of an instruction, i.e. the result of an operation exceeds the maximum number that can be written in the register. In case of no overflow, the condition code is set to 0, if the result equals 0; it is set to 1, if the result is less than 0 (negative); it is set to 2, if the result is greater than 0 (positive).

The instructions LOAD, STORE, MULTIPLY and DIVIDE have no effect on the condition code.

The condition code is used in the conditional branch instructions.

BRANCH ON CONDITION 47 BC RX 47  $M_1X_2B_2D_2$

In this instruction the field  $M_1$  (branch mask) is not a register address. The four mask bits correspond to the four possible values of condition code and determine the branch condition. If the condition is satisfied, a branch is made to address  $E_2$ , if not, the instruction next in sequence is obeyed.

If the programmer wants to make a branch at a certain value of the condition code (0, 1, 2 or 3), he must set the corresponding bit of the branch mask to 1. Thus,  $M_1 = 1000_2 = 8_{16}$  will cause a branch at CC = 0,  $M_1 = 0010_2 = 2_{16}$  will cause a branch at CC = 2, and  $M_1 = 1100_2 = C_{16}$  will cause a branch at CC = 0 or 1.  $M_1 = 1111_2 = F_{16}$  will cause a branch at any value of condition code, regardless of whatever the conditions may be. Such a branch is called *unconditional*. At  $M_1 = 0000_2 = 0_{16}$  no branch occurs at any condition code. At other than zero mask, the conditional branching instruction is designated NOP (no operation) which literally means 'branch under no condition'.

Prior to giving the program, note that both formulas have a common subexpression  $A^2 + B^2$ . It is natural to process it before testing the condition  $A > B$ . This is a common rule:

Before branching the computing process, it is advisable to carry out all actions common to all the branching operations.

In the previous programs the storage has to be allocated to variables and constants, while the instructions might be arranged anywhere in the storage. In this program the instructions too must be assigned definite storage location, since in the instruction BRANCH ON CONDITION the address of the instruction to which the branch is made should be known.

Allocate the storage as follows: the instructions, starting with address 004100, X—at address 004200, A—at address 004204, and B—at address 004208.

Let register 15 be the base register and suppose that it contains 004000

004100-03	L	58 1 0 F 204	$A \rightarrow R_1$	
004104-05	MR	1C 0 1	$(R_1) \times (R_1) \rightarrow R_1$	$(R_1) = A^2$
004106-09	L	58 3 0 F 208	$B \rightarrow R_3$	
00410A-08	MR	1C 2 3	$(R_3) \times (R_3) \rightarrow R_3$	$(R_3) = B^2$
000410C-0D	AR	1A 1 3	$(R_1) + (R_3) \rightarrow R_1$	$(R_1) = A^2 + B^2$
00410E-11	L	58 2 0 F 204	$A \rightarrow R_2$	
004102-15	S	5B 2 0 F 208	$(R_2) - B \rightarrow R_2$	$(R_2) = A - B$
004116-19	BC	47 C 0 F 122	$C_{16-1100_2}$	Branch at CC = 0 or 1
00411A-1D	S	5B 1 0 F 204	$(R_1) - A \rightarrow R_1$	$(R_1) = A^2 + B^2 - A$
00411E-21	BC	47 F 0 F 126	unconditional branch	
004122-25	A	5A 1 0 F 204	$(R_1) + A \rightarrow R_1$	$(R_1) = A^2 + B^2 + A$
004126-29	ST	50 1 0 F 200	$(R_1) \rightarrow X$	

Registers 0 through 3 are used by the program as working registers.

Note that in testing the condition  $A > B$ , we compute the difference  $A - B$  which is unnecessary. More than that, since the value of A in register 2 is lost, we shall have in future to take A from the storage.

To set the condition code use may be made of the following instructions:

```

COMPARE  59 C RX  R1X2B2D2
          (R1) - (E2)
COMPARE  19 CR  RR  R1R2
          (R1) - (R2)

```

Like in the SUBTRACT instruction the second operand is subtracted from the first operand, but the difference is written nowhere. The purpose of these instructions is to set the condition code. The condition code (CC) is set to 0, 1 or 2, if the difference is equal to 0, is negative, or is positive, respectively. Since the result is written nowhere, overflow will never occur, and CC will be never set to 3.

LOAD AND TEST 12 LTR RR  $R_1 R_2$   
( $R_2$ )  $\rightarrow R_1$

**LOAD COMPLEMENT**    13 LCR RR     $R_1 R_2$   
                                $\rightarrow (R_2) \rightarrow R_1$

LOAD POSITIVE 10 LPR RR  $R_1 R_2$   
 $[(R_2)] \rightarrow R_1$

LOAD NEGATIVE| 11 LNR RR  $R_1 R_2$   
 $-[(R_2)] \rightarrow R_1$

The instructions LTR, LCR, LPR, and LNR have no format R X. There is one more form of the conditional branch.

The instruction is the same as the conditional branching instruction BC 47, but the branch address is specified in 24 low-order bits of register  $R_2$ . Particular case is represented by general register 0: if  $R_2 = 0$ , no branch occurs regardless of the condition code. This is another form of the NOP instruction. Its mnemonic designation is NOPR.

1. Compile a program to find the largest of three numbers. Arrange the program and the numbers in the storage as you like. Load the largest number in register 0.

2. Construct an algorithm and compile a program to find the larger of two numbers A and B, using no conditional branching instruction.

3. Integer positive numbers A, B and C which stand for lengths of three line segments are stored in the main storage words at addresses 000A04, 000A08, and 000A0C. Write a program which will move to general register 12 one of the following codes:

- 0, if no triangle can be constructed from line segments,
- 1, if the triangle obtained is acute,
- 2, if the triangle obtained is right,
- 3, if the triangle obtained is obtuse.

#### 4.5. Programming Branching Algorithms

The techniques of programming branching algorithms is based on the conventional branching. It includes testing a logical condition and transfer to one of two program sections, depending on whether

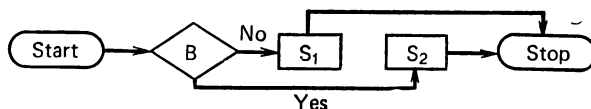


Fig. 4.2. Flowchart of program conventional branching  
 B—test to see whether logical condition (condition code formation) is true;  
 S<sub>1</sub> and S<sub>2</sub> are linear (without branches) sections of program

the condition is satisfied or not. After execution of each section control is transferred to the same instruction. A general flowchart of a conventional branch is shown in Fig. 4.2.

The arrow B → S<sub>2</sub> means a conditional branch. The arrow S<sub>1</sub> → Stop shows an unconditional branch. No special branching instructions are required to perform branches B → S<sub>1</sub> and S<sub>2</sub> → Stop, as the corresponding blocks can be written in a program directly in succession.

An example of conventional branching is the program given in the previous section. The reader is recommended to single out the instructions which refer directly to branching and to mark its constituents according to Fig. 4.2.

Programming conventional branches should not be difficult even for beginners. Note, however, the following recommendations which will help you to write better programs:

(a) Actions common to both branches of the program should be carried out before branching (look through the program in the previous section once more).

(b) Each instruction, except for COMPARE, setting the condition code also performs other actions, for which reason a most useful instruction must be utilized in each case. Consider a condition

$A > B$  as an example. If it is necessary to make an  $A - B$  (or  $B - A$ ) computation in the future, then to form a condition code use may be made of Subtract, otherwise, it is better to use Compare, the register being loaded by that of the operands which will be necessary in future. If  $A - B$  has been computed before, use may be made of LOAD and TEST. Versions like these are many, and to choose a best one depends upon the programmer's experience.

(c) None of the program characteristics will change, if sections  $S_1$  and  $S_2$  are exchanged, but in most cases, it is more convenient to program the more complicated section first.

A multiway branching contains several logical conditions and linear sections each of which is satisfied under a certain combination of conditions. The programming techniques for multiway branches are very miscellaneous and can be gained only practically. Therefore, it will be enough to illustrate it by an example.

An example below is far larger in scope than the previous problems. However, this should not frighten the reader. As has been shown by experience, the programming techniques can be better studied on examples close to actual problems, rather than on invented examples.

**Example.** An automatic traffic control system is planned to be installed in the town of N. The town streets are furnished with pick-

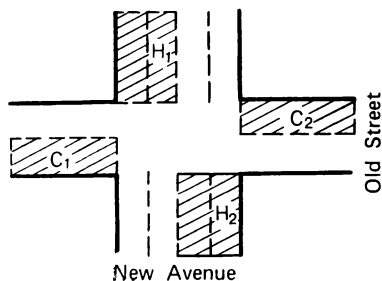


Fig. 4.3. Traffic diagram of street crossing

up units to count passing motor vehicles. The operation of the traffic lights should be under computer control.

Write a program to control the traffic lights at the crossing of the New Avenue and Old Street (Fig. 4.3).

Given are the number of motor vehicles ( $H_1$ ,  $H_2$ ,  $C_1$ , and  $C_2$ ) on the shaded areas which are counted by the pickup units. Our program is accessed once per a second. In that, the green

light direction should be determined for each second.

After an analysis of the traffic flows, the following traffic light control rules have been adopted.

1. All the time the Old Street is free from motor vehicles, the green light in the New Avenue is lighting.
2. A motor vehicle approaching the crossing along the Old Street should wait at the crossing for no more than 120 s.
3. A motor vehicle taking the New Avenue should be held at the crossing no more than 30 s.

4. The green light in the Old Street lights up for  $10 + 2C$  seconds where  $C = \max(C_1, C_2)$  is the greater of the numbers  $C_1$  and  $C_2$  the instant the traffic lights are changed.

5. The green light in the New Avenue is on  $10 + H$  seconds where  $H = \max(H_1, H_2)$  is the greater of the numbers the instant the traffic lights are changed.

6. The green light in the New Avenue will be lighting all the time, while though one of the numbers  $H_1$  or  $H_2$  is greater than 15, if this does not interfere with condition 2.

A reader who is able to answer whether the traffic light must be switched on and for what time may be proud of this. However, if there are still some confusions, it is good practice to read the problem statement once more, since we shall have to convert these vague rules into an unambiguous machine language.

You will easily note that in addition to the values of numbers  $H_1$ ,  $H_2$ ,  $C_1$ , and  $C_2$ , one should have some more information, namely, what is the traffic direction permitted by the green light, how much time will be taken to pass the motor vehicles crowded at the crossing, and what is the time motor vehicles are waiting to move in the other direction?

Let us use the following designations of the contents of the working locations:  $d$  is the direction in which the traffic is permitted:  $d = 0$ , if the traffic is permitted in the New Avenue, and  $d = 1$ , if it is permitted in the Old Street;  $t$  is the time elapsed after the traffic lights have been changed for the last time;  $t_p$  is the time taken to pass the motor vehicles crowded at the crossing;  $t_p = 10 + 2C$  seconds, if the traffic is permitted in the Old Street, and  $t_p = 10 + H$  seconds, if the traffic is allowed in the New Avenue;  $t_w$  is the waiting time taken by the motor vehicles stopped by the red light.

Suppose, that the values of  $d$ ,  $t$ ,  $t_p$ , and  $t_w$  for the first access to the program are specified externally. Then, the program will change these values in compliance with the situation at the crossing.

The value of  $d$  resulting from the program work controls the direction permitted by the green light. The values of  $t$ ,  $t_p$ , and  $t_w$  will be utilized in the subsequent accesses.

Now, let us define the precise instructions to control the traffic lights which will be later realized in a program. These instructions are given in the form of a flowchart vividly illustrating the algorithm (Fig. 4.4).

Before studying the flowchart, the reader is recommended to construct it by himself. Do not be afraid of this task and acknowledge defeat beforehand, because even an unsuccessful attempt is to the good: the problem and its difficulties will be clearer, and the reader will be in a position to better analyze the problem solution given in the book.

Block 1 counts the time elapsed after the last switching of the

traffic lights (recall, that the program is accessed once per a second).

Blocks 2, 3, and 4 count the values of  $C = \max(C_1, C_2)$ , blocks 5, 6 and 7, the value of  $H = \max(H_1, H_2)$ .

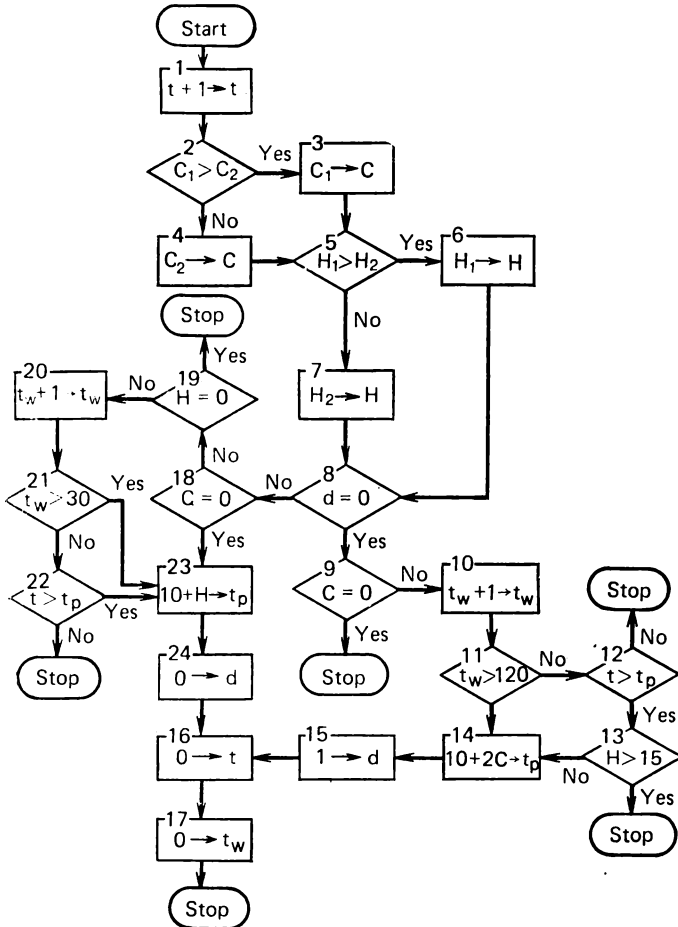


Fig. 4.4. Flowchart of traffic lights control algorithm

The direction of permitted traffic is determined in block 8. Depending on this, control is transferred to one of the two program sections. Blocks 9 through 17 determine the necessity of switching on the green light in the Old Street. The necessity of turning on the green light in the New Avenue is determined in blocks 18 through 24 and 16 through 17. These sections are similar, and detailed description of one of them will do.

Whether there are motor vehicles in the Old Street is determined in block 9. If not ( $C = 0$ ), then according to rule 1 the green light in the New Avenue must be lighting, and as it is actually lighting regardless of this ( $d = 0$ ), the program terminates its work.

If there are motor vehicles in the Old Street ( $C \neq 0$ ), block 10 adds 1 to the waiting time at the crossing, and block 11 tests to see whether it has not exceeded the critical value (2 min according to rule 2). At  $t_w > 120$  s, the traffic light in the Old Street must be set to green which is carried out by blocks 14 through 17.

At  $t_w \leq 120$  s, tests are made to see whether the motor vehicles have passed the New Avenue ( $t > t_p$  and  $H \leq 15$ ). If not so, the program terminates its work, otherwise the traffic light should be switched over, and control again transferred to block 14.

Writing a program following a prepared flowchart is not difficult, though there are details of importance in this matter. In writing, let the base value 004000 be in register 15, and the address of the instruction to which control should be transferred on exit from the program in register 14.

The instructions and constants will be arranged starting with address 004000, and the working fields, starting with address 004100. In studying the program, stress should be laid on masks of branch instructions.

004000-03	L	58 0 0 F 114	Block 1
004004-07	A	5A 0 0 F 0C0	$t + 1 \rightarrow t, R_0$
004008-0B	ST	50 0 0 F 114	
00400C-0F	L	58 1 0 F 108	Blocks 2, 3, 4
004010-13	C	59 1 0 F 10C	
004014-17	BC	47 A 0 F 01C	$\max(C_1, C_2) \rightarrow R_1$
004018-1B	L	58 1 0 F 10C	
00401C-1F	L	58 2 0 F 100	Blocks 5, 6, 7
004020-23	C	59 2 0 F 104	
004024-27	BC	47 A 0 F 02C	$\max(H_1, H_2) \rightarrow R_2$
004028-2B	L	58 2 0 F 104	
00402C-2F	L	58 3 0 F 110	Block 8
004030-31	LTR	12 3 3	Test $d = 0$
004032-35	BC	47 2 0 F 07A	
004036-37	LTR	12 1 1	Block 9
004038-39	BCR	07 8 E	Test $C = 0$
00403A-3D	L	58 3 0 F 11C	Block 10
00403E-41	A	5A 3 0 F 0C0	$t_w + 1 \rightarrow t_w, R_3$
004042-45	ST	50 3 0 F 11C	
004046-49	C	59 3 0 F 0CC	Block 11
00404A-4D	BC	47 2 0 F 05A	Test $t_w > 120$
00404E-51	C	59 0 0 F 118	Block 12
004052-53	BCR	07 C E	Test $t > t_p$

004054-57	C	59 2 0 F 0C8	Block 13
004058-59	BCR	07 2 E	Test $H > 15$
00405A-5D	L	58 3 0 F 0C4	Block 14
00405E-5F	AR	1A 3 1	
004060-61	AR	1A 3 1	$10 + 2C \rightarrow t_p$
004062-65	ST	50 3 0 F 118	
004066-69	L	58 3 0 F 0C0	Block 15
00406A-6D	ST	50 3 0 F 110	$1 \rightarrow d$
00406E-6F	SR	1B 3 3	Block 16
004070-73	ST	50 3 0 F 114	$0 \rightarrow t$
004074-77	ST	50 3 0 F 11C	Block 17 $0 \rightarrow t_w$
004078-79	BCR	07 F E	
00407A-7B	LTR	12 1 1	Block 18
00407C-7F	BC	47 8 0 F 09E	Test $C = 0$
004080-81	LTR	12 2 2	Block 19
004082-83	BCR	07 8 E	Test $H = 0$
004084-87	L	58 3 0 F 11C	Block 20
004088-8B	A	5A 3 0 F 0C0	$t_w + 1 \rightarrow t_w, R_3$
00408C-8F	ST	50 3 0 F 11C	
004090-93	C	59 3 0 F 0D0	Block 21
004094-97	BC	47 2 0 F 09E	Test $t_w > 30$
004098-9B	C	59 0 0 F 118	Block 22
00409C-9D	BCR	07 C E	Test $t > t_p$
00409E-A1	L	58 3 0 F 0C4	Block 23
0040A2-A3	AR	1A 3 2	$10 + H \rightarrow t_p$
0040A4-A7	ST	50 3 0 F 118	
0040A8-A9	SR	1B 3 3	Block 24
0040AA-AD	ST	50 3 0 F 110	$0 \rightarrow d$
0040AE-B1	BC	47 F 0 F 06E	Branch to block 16

*Constants*

0040C0-03	0000 0001	
0040C4-C7	0000 000A	$A_{16} = 10_{10}$
0040C8-CB	0000 000F	$F_{16} = 15_{10}$
0040CC-CF	0000 0078	$78_{16} = 120_{10}$
0040D0-D3	0000 001E	$1E_{16} = 30_{10}$

*Working Fields*

004100-03	$H_1$
004104-07	$H_2$
004108-0B	$C_1$
00410C-0F	$C_2$
004110-13	$d$
004114-17	$t$
004118-1B	$t_p$
00411C-1F	$t_w$

The value of  $t$  is stored in general register 0, the value of  $C$ , in register 1, and the value of  $H$ , in register 2. Register 3 is used by the program as the working one.

### Exercises

Continue the development of the traffic light control algorithm in one of the following aspects:

(a) In actual traffic control lights, changing over from green to red is preceded by yellow light. Introduce new variables to control the yellow light and formulate rules to enable and disable it.

(b) Only straight motion has been allowed at the crossing in question. Introduce new variables to control turn arrows permitting various turns. Work out rules to control the turn arrows to allow any turns, to prevent any accidents, and to make waiting periods at the crossing not too long.

(c) Take a crossing well known to you and work out the rules to control the traffic light so that the average waiting time of the motor vehicles is minimized as far as practicable.

Using the rules worked out, construct flowcharts and write programs. The arrangement of the program and data in the storage is any you like.

### 4.6. Subroutines. Linkage Conventions

One of the most important tools of the programmer is a subroutine. Sometimes, an analysis of the algorithm makes it possible to single out groups of statements that perform some desired procedure and obey similar instructions. If such groups are sufficiently long, then it is not advisable to write the same instructions each time when they are encountered. It is convenient to be able to write the common statements only once and use them as often as and wherever necessary. Such a block of common statements is called a *subroutine*.

The interaction between the main program and a subroutine is shown in Fig. 4.5. The solid-line arrows indicate references to the subroutine. Responses from the subroutine to the main program are shown by broken-line arrows.

It is seen from the diagram that the subroutine should know where-

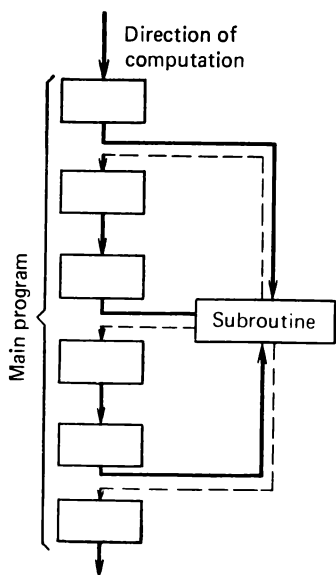


Fig. 4.5. Interaction between main program and subroutine.

from it is referenced in order to return to the required point of the main program. Therefore, in its reference the main program should transmit to the subroutine the address of the instruction to which control must be transferred on exit from the subroutine, or in short, the address of the return point. To this end, the computer uses special instructions:

```
BRANCH AND LINK  45  BAL  RX  R1X2B2D2
BRANCH AND LINK  05  BALR RR  R1R2
```

In the execution of this instruction the address of the subsequent instruction is stored in the three rightmost bytes of general register  $R_1$ . The branch is to the address of  $E_2$  for instruction BAL and to the address contained in the three rightmost bytes of register  $R_2$  for instruction BALR.

Again, particular case is with general register 0. If  $R_2$  in instruction BALR equals 0, no branch occurs. One of the possible uses of such an instruction will be considered below.

The left byte of the word stored in the general register  $R_1$  should not be regarded as containing zeros. It stores certain additional information on the status of the program. Instructions BAL and BALR will be considered in more detail in Sect. 5.3.

Write a program for computation of the polynomial

$$Y = (A^2 + 3A + 1) + (B^2 + 3B + 1) + (C^2 + 3C + 1)$$

It is readily seen that the program will contain three common places defining the parenthesized expressions.

Let the following computation be a subroutine

$$f(t) = t^2 + 3t + 1$$

The storage allocation will be as follows:

```
004000-03  Y
004004-07  A
004008-0B  B
00400C-0F  C
```

Write the main program starting with address 004020 and the subroutine, with address 004050. Let the value of base 004000 be in general register 12.

```
004020-23  L    58 2 0 C 004  A → R2
004024-27  BAL  45 E 0 C 050  Reference to subroutine
004028-29  LR   18 3 1         f(A) → R3
00402A-2D  L    58 2 0 C 008  B → R2
00402E-31  BAL  45 E 0 C 050  Reference to subroutine
004032-33  AR   1A 3 1         (R3) + f(B) → R3; (R3) = f(A) + f(B)
004034-37  L    58 2 0 C 00C  C → R2
```

004038-3B	BAL	45	E 0	C 050	Reference to subroutine
00403C-3D	AR	1A	3 1		$(R_3) + f(C) \rightarrow R_3; (R_3) = f(A) +$ $f(B) + f(C)$
00403E-41	ST	50	3 0	C 000	$(R_3) \rightarrow Y$
004050-53	L	58	1 0	C 060	$3 \rightarrow R_1$
004054-55	AR	1A	1 2		$(R_1) + t \rightarrow R_1; (R_1) = t + 3$
004056-57	MR	5C	0 2		$(R_1) \times t \rightarrow R_1; (R_1) = t^2 + 3t$
004058-5B	A	5A	1 0	C 064	$(R_1) + 1 \rightarrow R_1; (R_1) = t^2 + 3t + 1$
00405C-5D	BCR	07	F E		Exit from subroutine
004060-63		0000	0003		
004064-67		0000	0001		

The main program uses general register 3 for computation of  $Y$ , and register 2, for transmission of arguments to the subroutine. The subroutine places the computed value of  $f(t)$  in general register 1, and uses register 0 as the working register. General register 14 serves to organize the communication between the main program and subroutine.

The above-given program is very simple. Despite this, writing repeated segments in the form of a subroutine cuts down the number of instructions, compared with the direct programming. The instruction saving advantage rises with program complexity and/or more frequent references to the subroutine.

The reader can note that this saving in the program space leads to an increase in the program execution time, because in addition to the arithmetic instructions, branching instructions have to be obeyed to set links between the main program and subroutine. This is true not only of programming. Hardly it is possible to improve all characteristics at once. Usually, a compromise is made.

In this event it is advisable to make use of subroutines in all programs, except for the programs which must work as fast as possible. An increase in the operating time of a program due to the instructions organizing the linkage is minute, and is far paid for by the following advantages of using subroutines:

- A decrease, and often material, in the length of the program;
- Saving the programmer the trouble of rewriting instructions;
- Easier making of amendments in the program and easier debugging;
- Possibility of breaking a program into separate subprograms which can be compiled by several programmers, which is of importance in writing large programs.

Another, very likely, the most important use of subroutines is in utilizing *standard subroutines*. The practical experience of programmers has shown that algorithms of various computation processes usually contain specific problems that are frequently encountered. Examples are computations of square root, trigonometrical and

other functions, sets of algebraic equations, etc. All this has led to the idea that a programmer may code, say, a 'square root subroutine' and call that subroutine whenever a square root is needed in a program, instead of constantly repeating the coding of the steps over and over if square roots are needed often.

The programs for frequently encountered problems were called standard subroutines. Such subroutines were specially developed and accumulated. At present, libraries of standard subroutines for various computers contain many hundreds and thousands of such programs.

The wide application of standard subroutines has led to the necessity of introducing standards on programming in general.

Programs, standard subroutines included, are written by various people differing in experience and habits. If there were no certain common requirements imposed on writing programs, the result would be utter confusion: before an attempt to use any standard subroutine, the user would have to thoroughly investigate what computer resources (registers, addresses of storage, etc.) are utilized by the subroutine, what data are 'spoilt' by its use, what data and in what order are used by the subroutine as the source data, wherein the data must be stored, etc. The situation is still more aggravated by the fact that the information thus obtained would be useless for each new subroutine. This situation, by the way, was characteristic of the first generations of computers which often got in the way of utilizing standard subroutines and sometimes led to the necessity of recompiling, or materially modifying them.

The modern designers of computers take these requirements into account by implementing new facilities in the set of instructions of the computer and imposing certain requirements on any program. These, taken as a whole, are known as a *programming system*.

This section of the book deals with the basic requirements imposed on the standard subroutines and facilities provided by the ES EVM by means of which these requirements can be met.

**Relocatability.** One of the tasks to be carried out in writing a program is represented by arrangement of data and the program itself in the storage of the computer. When used, standard subroutines also must be assigned some storage space. In this event, the address of the main storage at which the programmer wishes (or is permitted) to store a standard subroutine often is other than the address at which it has been started in writing. If that is the case, the standard subroutine must be relocated after certain modification so that it is enabled to properly function in the other location.

One of the essential requirements for the standard subroutines is in that the adjustment to a location in the storage requires as little adjusting effort as possible. In the ideal case, the performance of the program is independent of the storage allocation. These programs are called *relocatable*.

Write, for instance, a program for computing  $X = A + B$ .  
Storage allocation

```
002010.13  X
002014.17  A
002018.1B   B
```

Let the program be started at address 002000 and the value of base 002000 be in register 10. Then, the program will be written as follows:

```
002000-03  L   58 1 0 A 014  A → R1
002004-07  A   5A 1 0 A 018  (R1) + B → R1
002008-0B  ST  50 1 0 A 010  (R1) → X
```

Now, assume that we are not satisfied by the above storage allocation, and we wish to start the program and data at addresses 002500 and 002510, respectively, i.e. to relocate them as a whole, with no changes in their relative positions. To do this, one method may be by changing the biases in the instructions by 514, 518, 510, respectively. However, it is readily seen, that it is best to change the content of the base register to 002500. It follows from this, that to make a program relocatable, it is enough to allow the program to define the address of its arrangement in the storage and to load it in the base register.

Such actions are performed by the BALR instruction in which  $R_1$  should be set equal to the base register, and  $R_2$  to zero. This instruction should precede all instructions of the program.

In this case the program will be as follows:

```
000-001  BALR  05 A 0j
002-005  L     58 1 0 A 012  A → R1
006-003  A     5A 1 0 A 016  (R1) + B → R1
00A-00D  ST    50 1 0 A 00E  (R1) → X
010-013  X
014-017  A
018-01B  B
```

The BALR instruction loads the address of the instruction that follows it in register 10. Therefore, the bias in all instructions is 2 less than the operand addresses indicated in the leftmost column.

As you see, the performance of this program is independent of the start address. Besides, you do not have to make a supposition on the value of the base register, as the program by itself sets the required base value.

Note also the addresses of the instructions and data. All the subsequent programs will be written by us in the relocatable form. Therefore, it is advisable to indicate only the bias of the instructions and data with regard to the beginning of the program.

**Saving.** The work of any program requires general registers. The content of some of these registers may vary with the execution of the program. The same registers may be utilized by the main storage, for which reason it is adopted in the ES EVM programming system that the content of the general registers at the exit from a subroutine must be the same as that at the instant the subroutine is referenced. This may not apply to the registers in which the results of program work are formed.

To save the content of the general registers, the main storage must hold those registers which will be changed with a view to restoring their previous values. If a subroutine utilizes several registers, then it is inconvenient to write instructions ST (STORE) and L (LOAD) to save each of these registers. The computer features special instructions to store and save the contents of several registers at once. These instructions have the RS format

STORE MULTIPLE	90 STM RS
90 $R_1 R_3 B_2 D_2$	$(R_1) \rightarrow E_2$
	$(R_1 + 1) \rightarrow E_2 + 4$
	.....
	$(R_3) \rightarrow E_2 + 4 (n - 1)$

where  $n$  is the number of registers written.

The contents of general registers  $R_1, R_1 + 1, \dots, R_3$  are written in sequential words of storage, starting with the word specified by the effective address  $E_2 = (B_2) + D_2$ . The registers are addressed in cycles, i.e. we regard the register with address 15 as followed by the register with address 0, then register with address 1, and so on. Therefore, if the instruction sets  $R_1$  greater than  $R_3$ , then loaded in the storage are the contents of registers  $R_1, R_1 + 1, \dots, 15, 0, 1, \dots, R_3$ . This instruction may be used to load the content of any number of registers from 1 to 16 in the storage.

**Example.** The instruction STM 90 2 5 0 800 loads the content of register 2 in the word at address 000800-03, that of register 3, at 000804-07, that of register 4, at 000808, and that of register 5, at 00080C-0F.

The instruction STM 90 E 1 0 AB0 loads the content of register 14 in the word at address 000AB0-B3, that of register 15, at 000AB4-B7, that of register 0, at 0000AB8-B8, and that of register 1, at 000ABC-BF.

LOAD MULTIPLE	98 LM RS $R_1 R_3 B_2 D_2$
	$(E_2) \rightarrow R_1$
	$(E_2) \rightarrow R_1 + 1$
	.....
	$(E_2) + 4 (n - 1) \rightarrow R_3$

where  $n$  is the number of registers loaded.

The LM instruction is opposite to the STM instruction. Loaded in sequence into general registers  $R_1, R_1 + 1, \dots, R_3$  are words starting with the effective address  $E_2$ .

**Linkage registers.** To standardize references to any subroutine the ES EVM has adopted certain conventions on use of the general registers to organize the communication between the main storage and a subroutine. Before attempting to access a program, the main program should load registers 0, 1, 13, 14, and 15 known as *linkage registers*, observing the following rules.

1. Registers 0 and 1 are intended for data exchange between the main program and subroutine. The main program must transfer to the subroutine the source data or parameters for the operation, and the latter must return to the main program the results of its performance. Often, only one or two parameters are transferred from the program to subroutine. Sometimes, the values that are to be transferred, are loaded directly in registers 0 and 1. The subroutine (function computation, for instance) may also return the result to one of those registers.

2. If the amount of data transferred from the main program to the subroutine (or the other way round) is too large, then direct transfer to registers 0 and 1 is impossible. If that is the case, then the *address of the main storage area* containing the data in question is loaded in register 1.

3. Register 15 must contain the address of the subroutine entry point. This address may be used by the subroutine as the base register.

4. Register 14 must contain the address of the main program return point.

5. Register 13 must contain the address of the save area in which the subroutine may store the contents of the general registers. No other function is performed by register 13.

The main program must provide a save area for the subroutine. Since the main program may have several subroutines, it would be unadvisable to reserve its own save area in each subroutine, because all subroutines can utilize in turn a common save area.

The save area consists of 18 words (72 bytes) and has the following structure:

Word	Function
1	Pointers. Used only in PL/I programming
2	Address of the save area which was used by the main program (the previous area)
3	Address of the save area specified by the subprogram for its subroutines (the subsequent area)
4	Address of the return point for saving the content of register 14

Word	Function
5	Address of the entry point for saving the content of register 15
6	For saving the content of register 0
7	For saving the content of register 1
...	...
18	For saving the content of register 12

The first word of the save area may be used by the programmer as he wishes, say, as a working field.

The second and third words of the save area are used in complicated programs. Let, for instance, main program A be making a reference to subprogram B which in turn has subroutine C. In this event there must be two save areas: the first area is allocated by program A to subprogram B (designate this area as  $S_B$ ), while the other area is offered by subprogram B to subroutine C (designate it as  $S_C$ ). Therefore, in reference to subprogram B, general register 13 should contain the address of  $S_B$ , and address of  $S_C$  when reference is made to subroutine C. That is, the function of the second and third words is to store various states of register 13. In our case, word 2 of the  $S_B$  area contains 0, as program A is not a subprogram of some other program, neither it has a save area, and word 3 of the  $S_B$  contains the address of  $S_C$ . Word 2 of the  $S_C$  contains the address of  $S_B$ , while word 3 of it contains 0, as subroutine C has no subroutines and does not specify a new save area. The techniques of manipulating register 13 and words 2 and 3 of the save area will be later illustrated by examples.

Words 4 through 18 of the save area are used by the subroutine to store the contents of the general registers, except for register 13.

Let us write the following computation program

$$Y = (A^2 + 3A + 1) + (B^2 + 3B + 1) + (C^2 + 3C + 1)$$

This program has been written above. Now, however, we shall write it in the relocatable form, obeying all the rules for documenting a subroutine.

#### Main Program

000-001	BALR	05 9 0	
002-005	L	58 D 0 9 036	Load the address of the save area in register 13
006-009	L	58F 0 9 03A	Load the address of the subroutine entry point in register 15
00A-00D	L	58 1 0 9 02A	$A \rightarrow R_1$
00E-00F	BALR	05 E F	Reference to subroutine
010-011	LR	18 3 0	$f(A) \rightarrow R_3$
012-015	L	58 1 0 9 02E	$B \rightarrow R_1$
016-017	BALR	05 E F	Reference to subroutine
018-019	AR	1A 3 0	$(R_3) + f(B) \rightarrow R_3$

01A-01D	L	58 1 0 9 032	$C \rightarrow R_1$
01E-01F	BALR	05 E F	Reference to subroutine
020-021	AR	1A 3 0	$(R_3) + f(C) \rightarrow R_3$
022-025	ST	50 3 0 9 026	$(R_3) \rightarrow Y$
026-027		0000	
028-02B	Y		
02C-02F	A		
030-033	B		
034-037			C
038-03B			Address constant: address of the save area
03C-03F			Address constant: address of the subroutine entry point
040-087			Save area for the subroutine of computing $t^2 + 3t + 1$
<i>Subroutine</i>			
000-003	STM	90 4 5 D 024	Save the contents of registers 4 and 5
004-007	L	58 5 0 F 018	$3 \rightarrow R_5$
008-009	AR	1A 5 1	$(R_5) + t \rightarrow R_5 \quad (R_5) = t + 3$
00A-00B	MR	1C 4 1	$(R_5) + t \rightarrow R_5 \quad (R_5) = t^2 + 3t$
00C-00F	A	5A 5 0 F 01C	$(R_5) + 1 \rightarrow R_5 \quad (R_5) = t^2 + 3t + 1$
010-011	LR	18 0 5	$(R_5) \rightarrow R_0$
012-015	LM	98 4 5 D	Recovery of the contents of registers 4 and 5
016-017	BCR	07 F E	Exit from the subroutine
018-01B		0000 0003	
01C-01F		0000 0001	

In the analysis of the above program note the following.

1. In order to load the address of the save area in register 13 (instruction 002) and the address of the subroutine entry point (instruction 006), the main program should have the address constants (038-03F). These words must contain the absolute addresses of the corresponding bytes. Therefore, a program having subroutines loses its relocatability: the value of address constants may be written in a free array of the main storage only after the program is related to an actual address of the main storage.

Unlike this, the subroutine of computing  $t^2 + 3t + 1$  is relocatable. A point to be noted is that in the ES EVM programming system a program should contain no irrelocatable elements, except address constants.

2. A thing to remember is that the fixed-point arithmetic instructions require that the operands in the main storage be started at the word boundary, i.e. with the byte whose address is a factor of 4, or, in short, be aligned with the 'word boundary'. To this end, it

is adopted that in relocating programs any subroutine must be arranged in the storage so that it starts at the double-word boundary. Therefore, in order to align any object with the word (halfword, double word) boundary, it is enough to make its bias with regard to the beginning of the program a factor of four (two, eight). In aligning with the boundary, there may be left blank bytes, as the case is with bytes 026-027 in our example.

3. The subroutine does not start with the instruction BALR, as the instant it is called for, the address of the program beginning is already in register 15. That is why, the biases in the subroutine instructions are equal to the addresses in the left-hand column, and two less in the main program.

4. The program is simple in structure, for which reason words 2 and 3 of the save area are not used.

The example that follows illustrates the techniques of handling the parameters, register 13 and words 2 and 3 of the save area.

Write a standard subroutine for computing

$$Y = (A^2 + 3A + 1) + (B^2 + 3B + 1) + (C^2 + 3C + 1)$$

The parameters of the subroutine are three numbers A, B and C. Load the result of the computation in general register 0. Since there are more than two parameters, they cannot be loaded in registers 0 and 1. So, suppose that numbers A, B and C occupy in sequence words in the main storage. The program calling our subroutine should load the address of the first byte of the argument field in register 1.

The subroutine of computing  $t^2 + 3t + 1$  remains unchanged and is neither given here. Only the main program is given below.

000-003	STM	90 E C D 00C	Save the contents of all registers, except the content of register 13
004-005	LR	18 9 F	Load the base register
006-009	L	58 2 0 9 040	Load the address of the save area
00A-00F	ST	50 2 0 D 008	Store the address of the next save area in word 3 of the current area
00E-011	ST	50 D 0 2 004	Store the address of the current save area in word 2 of the next save area
012-013	LR	18 D 2	Load the address of the next save area in register 13
014-017	L	58 F 0 9 044	Load the address of the subroutine entry point in register 15
018-019	LR	18 2 1	Address of the parameter field $\rightarrow R_2$
01A-01D	L	58 1 0 2 000	$A \rightarrow R_1$
01E-01F	BALR	05 E F	Reference to subroutine
020-021	LR	18 3 0	$f(A) \rightarrow R_3$

022-025	L	58 1 0 2 004	$B \rightarrow R_1$
026-027	BALR	05 E F	Reference to subroutine
028-029	AR	1A 3 0	$(R_3) + f(A) \rightarrow R_3$
02A-02D	L	58 1 0 2 008	$C \rightarrow R_1$
02E-02F	BALR	05 E F	Reference to subroutine
030-031	AR	1A 3 0	$(R_3) + f(C) \rightarrow R_3$
032-035	L	58 D 0 D 004	Restore the content of register 13
036-039	ST	50 3 0 D 014	$(R_3) \rightarrow$ save field of register 0
03A-03D	LM	98 E C D 00C	Restore the contents of the other registers
03E-03F	BCR	07 F E	Exit into an external storage
040-043	Address constant: the address of the save area		
044-047	Address constant: the address of the subroutine entry point		
048-08F	The save area for the subroutine of computing $t^2 + 3t + 1$		

Now we shall explain certain instructions of the program additionally:

- 000— the program changes the values of registers 0, 1, 2, 3, 9, 14, and 15. However, it is usual to save the contents of all registers in fairly complicated programs (this removes the concern for seeing whether the use of a register is 'legal').
- 006-013— these four instructions form a standard block of instructions to store the values of register 13, and to set a new address of the save area. Of course, any register may be used in place of register 2.
- 018— register 1 is for linkage to the subroutine of computing  $t^2 + 3t + 1$ , for which reason the address of the parameter field should be reloaded into any free register.
- 032— the standard instruction of restoring the content of register 13. The value is taken from word 2 of the save area in which it has been loaded by the 00E instruction.
- 036— the computed value of Y may not be directly loaded in register 0, as its previous value will be restored by the next instruction, for which reason the result of the program work is loaded in the save field of register 0.

### Exercises

1. Elaborate a method which makes it possible to do without address constants, and accordingly modify the program set forth in this section.
2. Document the traffic light control program given in the previous section as a standard subroutine. All the variables ( $H_1, H_2, C_1, C_2, d$ , etc.) must be transferred to the subroutine as parameters. Think over the most convenient format of transferring parameters.

### 4.7. Programming Loop Algorithms

In all the above-mentioned algorithms the computation process proceeded purposefully. All instructions, except some references to one and the same subroutine, were executed once. This state of things, as is shown by the programming practice, is an exception to the rule, rather than a common practice. Most of the algorithms contain returns to the program segments that have been processed.

An example is the widely known Euclid Algorithm for finding the Greatest Common Divisor (GCD) for two integer positive numbers  $m$  and  $n$ :

- Step 1.* If  $m$  is equal to  $n$ , go to step 4.
- Step 2.* If  $m$  is less than  $n$ , exchange their places.
- Step 3.* Assume  $m - n \rightarrow m$ . Go to step 1.
- Step 4.* Stop,  $m$  is the GCD.

Let us illustrate the work of this algorithm with the aid of real numbers. Let  $m = 18$  and  $n = 12$ . In this case, the computation is as follows.

- Step 1.*  $m = 18$ ,  $n = 12$ .  $m$  is not equal to  $n$ , go to step 2.
- Step 2.*  $m$  is greater than  $n$ , go to step 3.
- Step 3.* Let us assume  $m = 18 - 12 = 6$ . Go to step 1.
- Step 1.*  $m = 6$ ,  $n = 12$ .  $m$  is not equal to  $n$ , go to step 2.
- Step 2.*  $m$  is less than  $n$ . Let us assume  $m = 12$ ,  $n = 6$ . Go to step 3.
- Step 3.* Let us assume  $m = 12 - 6 = 6$ . Go to step 1.
- Step 1.*  $m = 6$ ,  $n = 6$ .  $m$  is equal to  $n$ , go to step 4.
- Step 4.* Stop. GCD (18, 12) = 6.

Steps 1, 2, and 3 of the Euclid Algorithm are repeated many times. Such algorithms are called *loop algorithms*, and the repeated segments are known as *loops*. Each repeating cycle of a loop is carried out with new values of certain variables participating in the computations. These variables are called *loop parameters*. In our case the  $m$  and  $n$  are loop parameters.

To organize loops, branch instructions are used in the program. There are various techniques of constructing looping programs. However, regardless of the loop organization technique, we can distinguish the preparatory part, loop body, modification of the parameters for new repetition of the loop, and test for the final value or condition.

The preparatory part of the loop sets the initial values of the parameters for the first iteration (repetition) of the loop. The special preparation may not be used, if the loop parameters already have the required values (an example is the result of an operation performed by the previous segment of the program).

The preparatory part of the loop, modification of the parameters, and testing the final result perform auxiliary functions: those are solely used to repeat the body of the loop the specified number of times at the required values of the loop parameters.

For a general flowchart of a looping segment of a program, see Fig. 4.6. The flowchart shown in Fig. 4.6 is most popular in constructing a loop. There are, however, loop algorithms for which this flowchart cannot be used. An example is the Euclid Algorithm. If  $m$

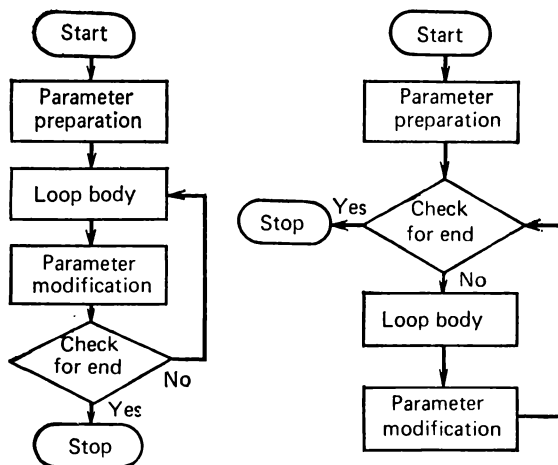


Fig. 4.6. General flowchart of looping segment

is set equal to  $n$ , no computations are needed, and it is necessary to exit the loop at once, not passing it at all. The looping programs in which the loop body may not be carried out at all should be constructed according to the flowchart shown in Fig. 4.6.

Described below are the principal types of loops: iteration loops with a counter, loops with readdressing. The methods of their organization in the program are also dealt with.

**The iteration loops** are characterized by the fact that the computations are performed each time by the same formulas, the computation result being utilized as the source data in the following repetition (iteration) of the loop. An example of this loop is the Euclid Algorithm. To test the termination of such loops, use is generally made of the conditional branch instruction.

Let us write a standard subroutine for computing the GCD for two integer positive numbers according to the Euclid Algorithm. The parameters are transferred to the subroutine in the linkage registers:  $m$  in general register 0,  $n$  in register 1.

The computation outcome is to be loaded in general register 0.

The program for computing the GCD of two integer positive numbers according to the Euclid Algorithm is as follows

000-003	ST	50 2 0 D 01C	Save the contents of register 2
004-005	CR	19 0 1	Compare $(R_0) - (R_1)$ to zero
006-009	BC	47 8 0 F 01A	Conditional branch at $CC = 0$
00A-00D	BC	47 2 0 F 014	Conditional branch at $CC = 2$
00E-00F	LR	18 2 0	$(R_0) \rightarrow R_2; (R_2) = m$
010-011	LR	18 0 1	$(R_1) \rightarrow R_0; (R_0) = n$
012-013	LR	18 1 2	$(R_2) \rightarrow R_1; (R_1) = m$
014-015	SR	1B 0 1	$(R_0) - (R_1) \rightarrow R_0; (R_0) = m - n$
016-019	BC	47 F 0 F 004	Unconditional branch to step 1
01A-01D	L	58 2 0 D 01C	Restoration of the contents of register 2
01E-01F	BCR	07 F E	Exit from the subroutine

No preparatory part is used in this program, since the parameters  $m$  and  $n$  have been loaded in the registers by an external program. Instructions 004 and 006 test the loop termination. With the condition  $m = n$  satisfied, instructions 01A and 01E will operate to organize exit from the subroutine. The next instruction is a conditional branch at  $m > n$ . The previous instruction BC has no effect on the condition code, and the instruction COMPARE may not be written again. Instructions 00E, 010, and 012 exchange places of  $m$  and  $n$ . Note that this exchange cannot be accomplished by the instructions

LR 18 0 1  
LR 18 1 0

as the former one erases the content of register 0. As a result, both registers will contain the same number. Therefore, use should be made of an additional register. Instruction 014 modifies the value of parameter  $m$ , and the next instruction transfers control to the beginning of the loop. The source data are not saved during the program work.

The loops with counter are characterized by the fact that their number of iterations is known, for which reason testing the loop for termination can be as follows. During the preparatory part, the number of iterations is entered in the general register known as a counter, and the loop is controlled by 1-decrementing the counter on every iteration (cycle) until its value is reduced to zero, after which an exit from the loop occurs.

Let us write a standard subprogram for computing the value of  $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n$  for integer positive  $n$ .

It is supposed that the value of  $n$  is such that  $n!$  can be stored in one register. Parameter  $n$  is transferred to the subprogram, into

linkage register 1. Store the results of computation in general register 0. The program is given below.

000-003	STM	90 2 4 D 01C	Save the contents of registers 2-4
004-007	L	58 3 0 F 01C	$1 \rightarrow R_3$
008-009	LR	18 4 1	$n \rightarrow R_4$ , setting counter
00A-00B	MR	1C 2 4	$(R_3) \times \text{counter} \rightarrow R_3$
00C-00F	S	5B 4 0 F 01C	Counter content $1 \rightarrow \text{counter}$
010-013	BC	47 2 0 F 00A	Branch at counter content $\neq 0$
014-015	LR	18 0 3	Result $\rightarrow R_0$
016-019	LM	98 2 4 D 01C	Restoration of the contents of registers 2-4
01A-01B	BCR	07 F E	Exit from the subprogram
01C-01F		0000 0001	

In writing the program we have utilized the fact that

$$n! = n(n-1) \dots 2 \cdot 1$$

Therefore, the content of the counter can be used also as the loop parameter. This technique is characteristic of loops with counters: often, it is more convenient to carry out computations with subtracting one from the counter on every cycle. The program utilizes register 3 for computing the product and register 4 as the counter. Register 2 is used as working in the MULTIPLY instruction.

The loops with counters are so often encountered in various algorithms that to make their programming easier, the computer has special instructions.

BRANCH ON COUNT 46 BCT RX 46  $R_1 X_2 B_2 D_2$   $(R_1) - 1 \rightarrow R_1$ .  
At  $(R_1) \neq 0$  branch to  $E_2$ . BRANCH ON COUNT 06 BCTR RR  
06  $R_1 R_2$   $(R_1) - 1 \rightarrow R_1$ . At  $(R_1) \neq 0$  branch to  $(R_2)$

These instructions as if combine operations of checking the loop termination which was carried out by two instructions: SUBTRACT-ION and CONDITIONAL BRANCH. The content of register  $R_1$  is automatically decremented by one on each cycle, until its value is reduced to zero after which the next instruction is carried out. Otherwise, a branch occurs to effective address  $E_2$ , in case of a BCT instruction, and to the address specified by the three low-order bytes of register  $R_2$ , in case of a BCTR instruction. Like in the previous BRANCH instructions of format RR, whatever the content of the counter, no branch occurs, if in the BCTR instruction  $R_2 = 0$ . Such an instruction can be used to reduce the content of the counter without branch and generally to subtract one from a positive number contained in the register.

[By means of the instruction **BRANCH ON COUNT** the previous example can be programmed as follows:

000-003	STM	90 2 4 D	01C	Save the contents of registers 2-4
004-005	LR	18 3 1		$n \rightarrow R_3$
006-007	LR	18 4 1		$n \rightarrow R_4$ , set counter
008-00B	BCT	46 4 0 F	014	Check loop termination
00C-00D	LR	18 0 3		Result $\rightarrow R_0$
00E-011	LM	98 2 4 D	01C	Save the contents of registers 2-4
012-013	BCR	07 F E		Exit from the subprogram
014-015	MR	1C 2 4		$(R_3) \times \text{content of counter} \rightarrow R_3$
016-019	BC	47 F 0 F	008	Branch to start of loop

In this program, in contrast to the previous one, we load  $n$  in register 3, rather than one. Thus, we save one multiply instruction. However, at  $n = 1$  the program work must terminate, for which reason the test for loop termination should be made before the execute part of the cycle.

In all the previous examples each iteration involved computations with different values of variables, while the addresses of the variables remained unchanged.

**Loops with address modification** are used when each new iteration calls for handling the content of different (subsequent, for instance) storage locations. Therefore, the addresses of loop parameters (variables) must be modified at each loop iteration.

The addresses may be modified by varying the content of the register underlying the loop parameters, but generally, it is more convenient to make use of the index register.

We now write a standard subprogram of computing the sum of 100 numbers occupying consecutive words in the main storage, the address of the first of them being in general register 1, while the result of computations must be stored by the program in general register 0. The program is written below.

000-003	STM	90 2 3 D	01C	Save the contents of registers 2 and 3
004-005	SR	1B 0 0		Clear the content of register 0 for sum
006-007	SR	1B 3 3		Clear register 3 for index
008-00B	L	58 2 0 F	020	$100 \rightarrow R_2$ , set counter
00C-00F	A	5A 0 3 1	000	$(R_0) + E_2 \rightarrow R_0$
010-013	A	5A 3 0 F	024	$(R_3) + 4 \rightarrow R_3$ , increase index
014-017	BCT	46 2 0 F	00C	Test for loop termination
018-01B	LM	98 2 3 D	01C	Restore the contents of registers 2 and 3
01C-01D	BCR	07 F E		Exit from the subprogram
01E-01F		0000		
020-023		0000 0064		Counter $64_{16} = 100_{10}$
024-027		0000 0004		Index increment

During the first iteration of loop index register 3 will contain zero. The ADD instruction (00C) will add the first number to register 0. In the subsequent iterations of loop index register 3, and therefore, the effective address in the instruction ADD will be incremented by 4 (word length) and the next numbers will be added to the sum. Register 0 is first cleared by instruction 004 (one of the instructions of the preparatory part of the loop).

In programming loops with address modification, changes of the index register often may be combined with test for loop termination, without using a register for the counter. An example is one more version of the program for adding 100 numbers.

000-003	ST	50 3 0 D	020	Save the content of register 3
004-005	SR	1B 0 0		Clear register 0 for the sum
006-009	L	58 3 0 F	01C	Load the initial value of index
00A-00D	A	5A 0 3 1	000	Add the subsequent number
00E-011	S	5B 3 0 F	020	Modify the index
012-015	BC	47 A 0 F	00A	Test for loop termination
016-019	L	58 3 0 D	020	Save the content of register 3
01A-01B	BCR	08 F E		Exit from the subprogram
01C-01F		0000 018C		$18C_{16} = 396_{10}$
020-023		0000 0004		

Note that with this organization of the loop, it is more convenient to add the numbers in the reverse order (why?). The initial content of index register 3 is set in this event  $396 = 4 \times (100 - 1)$ , so that in the first execution of the ADD instruction, the last (100th) number will be added to register 0 first cleared by the instruction SR. Then, four is subtracted from the content of index register 3, i.e. in the next iteration of the loop, the last but one number will be added to the sum, and so on. The instruction BRANCH ON CONDITION (BC) with a mask equal to  $A_{16} = 1010_2$  is executed at once after four has been subtracted from the index. This instruction utilizes the CONDITION CODE (CC) after the instruction SUBTRACT. At  $CC = 0$ , or  $CC = 2$ , i.e. while the index in register 3 is greater than or equal to 0, branch to the beginning of the loop will be executed, otherwise the loop will be terminated.

To facilitate organization of loops with address modification like this, the computer has two special instructions of the RS format which are as if a combination of instructions ADD, COMPARE and BRANCH ON CONDITION:

BRANCH ON INDEX LOW OR EQUAL	87	BXLE	RS	$R_1R_3B_2D_2$
BRANCH ON INDEX HIGH	86	BXH	RS	$R_1R_3B_2D_2$

These two instructions are executed as follows.

1. The contents of registers  $R_1$  and  $R_3$  are added and the result is loaded to register  $R_1$  replacing its previous content.

2. The new content of register  $R_1$  is compared to the content of register  $R_3$  or  $R_3 + 1$  (an odd-numbered register is selected).

3. Branch to the effective address  $E_2 = (B_2) + D_2$  occurs if  $(R_1) \leq (R_3 \text{ or } R_3 + 1)$  in case of instruction BXLE, or  $(R_1) > (R_3 \text{ or } R_3 + 1)$  when use is made of instruction BXH.

Register  $R_1$  contains the first operand (the current index value) which is incremented in the execution of the instruction. The *increment*, i.e. a value the first operand is increased by, is loaded in register  $R_3$ . This value may be either positive (increment), or negative (decrement). The limit value to which the value of the index is compared should also be in a general register, but its number is not explicitly specified in the instruction. The limit value should always be in an odd register and its number is determined by the following rule:

If register  $R_3$  is even-numbered, the limit value should be in the neighbouring odd-numbered register  $R_3 + 1$ .

If register  $R_3$  is odd-numbered, then it is considered that an odd register is used in the instruction for the increment and limit value.

It will be good to describe one more instruction here which is not included by any group of instructions, but is useful for certain auxiliary operations:

LOAD ADDRESS    41   LA   RX      $R_1 X_2 B_2 D_2$   
     $E_2 \rightarrow R_1$

In the execution of this instruction the effective (actual) address  $E_2 = (X_2) + (B_2) + D_2$  is placed in general register  $R_1$ . The eight high-order bits (the leftmost byte) of register  $R_1$  are zero-filled. Note that in register  $R_1$  it is exactly the *effective address*  $E_2$  that is loaded, rather than the content of the storage area with this address. There is no access to the memory in the execution of the LA instruction. The computation of the  $E_2$  value loaded in the register is carried out in compliance with all rules for address computation, proceeding from the contents of the base register, index register, and displacement.

Some possible examples of using the load address (LA) instruction are as follows:

1. Loading a positive number not in excess of  $FFF_{16} = 4095_{10}$  in a general register. For example, the instruction

LA    41 9 0 0 028

loads number 0000 0028 in register 9.

2. Increasing the content of a general register by a number not in excess of 4095 with writing in the same or another register. Thus, the instruction

LA    41 2 0 2 004

adds four to the content of register 2, while the instruction

LA 41 5 0 4 001

increments the content of register 4 by 1, and places the result in register 5. Using the LA instruction to this end, the programmer must see to that the result of addition is positive and fits 24 bits, otherwise the high-order bits of the sum will be truncated and the operation result will be invalid.

3. Use of the LA instruction in place of address constants.

Given below is the third example of finding the sum of 100 numbers with use of instructions BXLE and LA.

000-003	STM	90 3 5 D 020	Save the contents of registers 3 and 5
004-005	SR	1B 0 0	Clear register 0 for the sum
006-007	SR	1B 3 3	Clear register 3 for the index
008-00B	LA	41 4 0 0 004	Increment (4) → R <sub>4</sub>
00C-00F	LA	41 5 0 0 18C	Limit value (396) → R <sub>5</sub>
010-013	A	5A 0 3 1 000	Add the immediate number
014-017	BXLE	87 3 4 F 010	Test for loop termination
018-01B	LM	98 3 5 D 020	Restore the contents of registers 3 and 5
01C-01D	BCR	07 F E	Exit from the subprogram

This program takes six bytes of storage less than the previous version. Its operating time is also less, since a less number of instructions (A and BXLE) is executed in the loop iteration. The LOAD ADDRESS (LA) instructions are used for loading the index increment in register 4 and limit value in register 5.

A patient reader can study the fourth (last) version of loop organization which utilizes the BXH instruction.

000-003	STM	90 2 3 D 01C	Save the contents of registers 2 and 3
004-005	SR	1B 0 0	Clear register 0 for the sum
006-009	LM	98 2 3 F 018	396 → R <sub>2</sub> ; -4 → R <sub>3</sub>
00A-00D	A	5A 0 2 1 000	Add the immediate number
00E-011	BXH	86 2 3 F 00A	Test for loop termination
012-015	LM	98 2 3 D 01C	Restore the contents of registers 2 and 3
016-017	BCR	07 F E	Exit from the subprogram
018-01B		0000 018C	
01C-01F		FFFF FFFC	

In this example the numbers are again added in the reverse order, and the index increment is negative and equal to four. In the BXH instruction one general register 3 is used for the increment and limit value of the index. It should be also noted that the LM instruction is concurrently used for loading the index initial value equal to 396 in register 2 and index increment equal to -4, in register 3.

Generally the programs contain different types of loops interlinked in diverse ways. There are programs with independent loops and with nested loops in which one is contained within another. Building a program using sequencing independent loops causes no difficulties compared with the above examples. Programming nested loops often calls for much ingenuity.

The nested loop is called an *inner* loop with regard to the loop it is included in, while the nesting loop is termed an *outer* loop relative to the former one. Inner loops in turn can have their inner loops. Each of loops may be of any of the above-considered types. In building programs containing nested loops, a point to watch is that during each entry in an inner loop its preparatory part must be executed.

Generally, most difficulties are encountered in programming nested loops with address modification. The program given below illustrates the technique of handling indices in nested loops.

Let us write a subprogram for transposing a square matrix  $n \times n$ . The elements of the original matrix are stored in rows, i.e. in the following sequence

$$a_{11}, a_{12}, \dots, a_{1n}, a_{21}, a_{22}, \dots, a_{2n}, \dots, a_{n1}, a_{n2}, \dots, a_{nn}$$

The rows of the transposed matrix correspond to the columns of the original matrix:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \rightarrow \begin{pmatrix} a_{11} & a_{21} & \dots & a_{n1} \\ a_{12} & a_{22} & \dots & a_{n2} \\ \dots & \dots & \dots & \dots \\ a_{1n} & a_{2n} & \dots & a_{nn} \end{pmatrix}$$

The program work results in a transposed matrix in the same memory area as the original one. The parameters are transferred to the subprogram in the linkage registers: the externally stored program places the order of the matrix in the form of an integer  $n$  in general register 0, and the initial address of the memory area occupied by the matrix, in register 1.

The flowchart of the algorithm is shown in Fig. 4.7. The algorithm includes two loops nested in each other. The inner loop with parameter  $j$  exchanges the elements of the  $i$ th row which are found to the right of the principal (main) diagonal of the matrix with the elements of the  $i$ th column which are situated below the principal diagonal. The outer loop with parameter  $i$  is intended for handling the rows from 1 to  $n$ .

The flowchart of the algorithm is fairly simple, mainly because the subscripts of the matrix elements are written in the mathematical form. However, in building the program one has to work hard on forming the required values of the subscripts in the registers.

Programming of a nested loop should be started with the body of the very inner loop. In our problem the body of the very inner loop is represented by symbols 5 and 6 exchanging elements  $a_{ij}$  and  $a_{ji}$ . Considering the required values to be in subscript registers, say, registers 2 and 3, the exchange can be effected through four instructions:

L	58	A	2	1	000	$a_{ij} \rightarrow R_{10}$
L	58	B	3	1	000	$a_{ji} \rightarrow R_{11}$
ST	50	A	3	1	000	$(R_{10}) \rightarrow a_{ji}$
ST	50	B	2	1	000	$(R_{11}) \rightarrow a_{ij}$

Therefore, in the next iteration of the inner loop register 2 must contain the displacement of element  $a_{ij}$  with regard to the beginning of the area occupied by the matrix which equals  $4[n(i-1) + j - 1]$ , and the displacement of element  $a_{ji}$  equal to  $4[n(j-1) + i - 1]$  in register 3.

Referring to the formulae for displacements, the increment of the subscript in register 2 with  $j$  incremented by 1 is equal to 4, while the increment of the subscript in register 3 equals  $4n$ . To increment the subscripts assign register 4 to number 4 and register 7 to number  $4n$ , respectively.

Now, we must solve the problem of testing for termination of the inner loop. We shall control the completion of the inner loop by the value of the subscript in register 2. Limit value  $j$  is equal to  $n$ , for which reason the inner loop should be continued until the value of the subscript in register 2 becomes greater than the displacement of element  $a_{in}$  equal to  $4[n(i-1) + n - 1] = 4(ni - 1)$ .

The limit value of the subscript of the inner loop should be assigned register 5 neighboured by the register containing the increment.

As can be seen from the flowchart, the original value of the subscripts in registers 2 and 3 before entering the inner loop should equal the displacement of diagonal element  $a_{ji}$ , i.e.  $4[n(i-1) + i - 1] =$

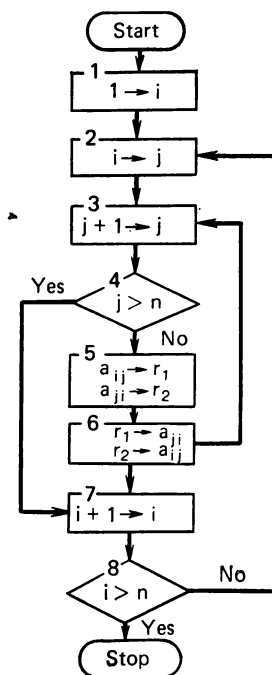


Fig. 4.7. Flowchart of matrix transposition algorithm

$= 4(n+1)(i-1)$ . Assign the original value of the subscripts of the inner loop register 6 and let it be moved in the preparatory part of the inner loop to registers 2 and 3.

The train of thought in accomplishing the outer loop is exactly the same. Incrementing  $i$  by one increases the value of the subscript in register 5 equal to  $4(ni-1)$  for  $4n$  and the value of the subscript in register 6, for  $4(n+1)$ . The new increment  $4(n+1)$  is assigned register 8.

Testing for the termination of the outer loop will be made according to the change in the subscript in register 6. The limit value of  $i$  is equal to  $n$ . Therefore, the limit value of the subscript in register 6 is equal to  $4(n^2-1)$ . It should be assigned register 9 neighbored by the register containing the corresponding increment.

The original value of register 6 (at  $i=1$ ) equals 0, and that of register 8,  $4(n-1)$ .

For convenience let us tabulate the results of our considerations into a table of utilizing the registers (Table 4.1).

Table 4.1

Register	Value	Explanation
2	$4[n(i-1) + j-1]$	Control of inner loop termination. The original value is $4(n+1)(i-1)$ . The limit value is $4(ni-1)$ . The increment is 4.
3	$4[n(j-1) + i-1]$	Utilized in the inner loop. The original value is $4(n+1)(i-1)$ . The increment is $4n$ .
4	4	Constant. The increment of the subscript is in register 2.
5	$4(ni-1)$	The limit value of the inner loop subscript. The original value is $4(n-1)$ . The increment is $4n$ .
6	$4(n+1) \times (i-1)$	Control of the outer loop termination. The original value is 0. The limit value is $4(n^2-1)$ . The increment is $4(n+1)$ .
7	$4n$	Constant. The increment of the subscripts in registers 3 and 5.
8	$4(n+1)$	Constant. The increment of the subscript in register 6.
9	$4(n^2-1)$	Constant. The limit value of the subscript in register 6.

In addition to the registers tabulated, the program uses registers 10 and 11 as working registers in the body of the inner loop.

It is good practice to draw up tables like this before writing instructions realizing any fairly intricate segment of a program. If

not so, disorderly assignment of registers will involve superfluous data transfers to the registers with necessary parity which is required by many instructions. Examples are MULTIPLY, BRANCH ON INDEX, etc.

The subprogram for transposing a matrix is given below.

000-003	STM	90 2 B D 01C	Save the contents of registers 2 through 11
004-007	LA	41 4 0 0 004	$4 \rightarrow R_4$
008-009	LR	18 7 0	$n \rightarrow R_7$
00A-00B	MR	1C 6 4	$(R_7) \times (R_4) \rightarrow (R_6, R_7); (R_6) = 0;$ $(R_7) = 4n$
00C-00D	LR	18 9 0	
00E-00F	MR	1C 8 7	
010-011	SR	1B 9 4	$4(n^2 - 1) \rightarrow R_9$
012-015	LA	41 8 0 7 004	$4(n + 1) \rightarrow R_8$
016-017	LR	18 5 7	
018-019	SR	1B 5 4	$4(n - 1) \rightarrow R_5$
01A-01B	LR	18 2 6	Beginning of outer loop
01C-01D	LR	18 3 6	
01E-01F	AR	1A 3 7	Beginning of inner loop
020-023	BXLE	87 2 4 F 030	Test for termination of inner loop
024-025	AR	1A 5 7	
026-029	BXLE	87 6 8 F 01A	Test for termination of outer loop
02A-02D	LM	98 2 B D 01C	Restore the contents of registers 2 through 11
02E-02F	BCR	07 F E	Exit from the subprogram
030-033	L	58 A 2 1 000	$a_{ij} \rightarrow R_{10}$
034-037	L	58 B 3 1 000	$a_{ji} \rightarrow R_{11}$
038-03B	ST	50 A 3 1 000	$(R_{10}) \rightarrow a_{ji}$
03C-03F	ST	50 B 2 1 000	$(R_{11}) \rightarrow a_{ij}$
040-043	BC	47 F 0 F 01E	Branch to the beginning of inner loop

The program consists of two successive sections. The first of them (instructions 004 through 019) is preparatory. It forms necessary constants and original values of subscripts in the registers. The other section starts with the instruction 01A corresponding to symbol 2 in the flowchart and represents a double loop transposing the matrix.

This book section completes the description of general techniques for programming. The branching, subprogram and loop are principal 'bricks' suitable for building any program. The further mastering of the proficiency in programming should take two directions.

1. The study of computer science theory, i.e. mastering the whole of the varieties a programmer is offered now by the developers of modern computers and their software. The above most simple program of adding 100 numbers (items) may have given the reader con-

fidence that even few studied instructions may be used to write a fairly large number of program versions. Still more methods can be used to write a matrix transposition program, and the authors were in difficulty as to what method was better to use in the text.

Generally, the main difficulty in building a program often lies in selecting a method best suitable for the purpose. Therefore, a skilled programmer should, without fail, be fully conversant with all abilities of the system, its advantages and disadvantages which enables him to arrive at a best decision in each practical case without obeying his own settled practice and inclinations.

2. Mastering practical skills of programming, since it is difficult, if not feasible at all, to learn a subject solely by reading the theory without using the knowledge acquired for tackling problems and thus making yourself consider what has been read again and again. To help the reader each section of this publication is furnished with exercises.

### Exercises

1. Modify the Euclid Algorithm for computing the GCD of two numbers so that there is no need to exchange the numbers in step 2. Rewrite the program respectively.

2. Prepare a standard subprogram for presenting integer numbers in factor form. The integer in linkage register 1 is transferred to the subprogram as a parameter. The result of the subprogram work should be the least divisor of the original number placed in register 0. Should the original number be a prime number, load register 0 with one.

3. Proceeding from exercise 2, approximately evaluate the working time of your program depending upon the value of the original number and value of its GCD (see Exercise 5 to Section 2 of this chapter). In addition, consider the execution time of BRANCH ON CONDITION instructions as equal to  $t$ . The quality of a program written by you is solely a function of its working time. Do your best to reduce this time as far as possible.

4. The memory contains a bulk of integers. Devise an algorithm and write a program for sorting these integers in order from lowest to highest. The quantity of numbers in the bulk and the initial address of the storage area occupied by the bulk of integers are given to your program as parameters. The result should be an array of numbers in order from lowest to highest in the same storage area.

5. A bus ticket number comprised by six digits is considered to be 'lucky', if the sum of its first three digits equals that of the last three digits. Devise an algorithm and write a program for defining the number of various "lucky" numbers.

Usually fixed-point instructions perform operations on fullwords and allow processing of numbers ranging from  $-2,147,483,648$  to  $+2,147,483,647$ . In many problems numbers are encountered which contain far less digits, and it is uneconomical to store them in the form of fullwords. Therefore, the computer provides the possibility of storing small integer numbers (from  $-32,748$  to  $+32,747$ ) in the form of halfwords. To handle such numbers, use should be made of special instructions. There are six instructions in the computer which handle halfwords. These are:

LOAD HALFWORD	48	LH	RX
COMPARE HALFWORD	49	CH	RX
ADD HALFWORD	4A	AH	RX
SUBTRACT HALFWORD	4B	SH	RX

These instructions are halfword equivalents of the conventional instructions L, C, A and S. The result of their work is exactly the same, as if a fullword of storage were occupied by the second operand and completely. First the halfword is expanded to a fullword, and then the operation is carried out as usually.

The instruction functions in a manner similar to the instruction ST, but stored is only the right half of the general register defined by the first operand.

MULTIPLY HALFWORD      4C   MH   RX       $R_1X_2B_2D_2$   
     $(R_1) \times (E_2) \rightarrow R_1$

Unlike the conventional instruction MULTIPLY, the instruction MULTIPLY HALFWORD (MH) utilizes only one register which may be of any parity. The content of register R<sub>1</sub> is multiplied by the content of the halfword in storage, and the result is stored in place of the first operand. The MH instruction is for multiplying two small numbers, and it is supposed that the result will fit the register. If the register R<sub>1</sub> contains a large number, and the product will be too large to fit a 32-bit register, then only the 32 rightmost bits of the product will be saved. There is no indication to the programmer when this occurs, i.e. no overflow is indicated during the execution of the MH instruction.

Another expansion of a conventional set of fixed-point arithmetical instructions is represented by tools of processing large numbers which do not fit a register.

Examples of such numbers are double-precision numbers resulting from execution of the MULTIPLY instruction. Double-precision computations are used in many problems, for instance, in the com-

putation of tables containing values of various functions with many significant digits. Long numbers may occupy in storage two, three and more words, depending upon the problem requirements. The computer has no instructions to process such numbers and arithmetic operations have to be performed in sequence on each word containing part of a given number.

The sign of a long number is contained only in the first of the words it occupies, while the sign bits of the other words are conventional digits of the number. Therefore, arithmetic operations on the second and subsequent words of the given number cannot be performed with the aid of conventional fixed-point arithmetic instructions. To this end use is made of special instructions:

ADD LOGICAL	1E ALR RR $R_1R_2$
	$(R_1) + (R_2) \rightarrow R_1$
ADD LOGICAL	5E AL RX $R_1X_2B_2D_2$
	$(R_1) + (E_2) \rightarrow R_1$

In the execution of these instructions all the 32 bits of a word, the sign bit inclusive, are treated as the digits of the number. In a special manner the ADD LOGICAL instructions set the condition code which allows it to be determined whether the sum fits the register, i.e. whether a carry from the sign bit has occurred. The condition code is set as follows:

- 0— zero sum; no carry from the sign bit; this situation can take place solely when both summands are equal to zero;
- 1— non-zero sum; no carry from the sign bit occurs;
- 2— zero sum; a carry from the sign bit occurred;
- 3— non-zero sum; a carry from the sign bit occurred.

### Example

#### *Before instruction execution*

Register 2	12 34 5678
Register 3	87 65 4321
Register 4	ED CB A988

#### *Instruction*

#### *After instruction execution*

ALR 1E 2 3	Register 2	9999 9999 CC = 1
ALR 1E 2 4	Register 2 <span style="border: 1px solid black; padding: 2px;">1</span>	0000 0000 CC = 2
ALR 1E 3 4	Register 3 <span style="border: 1px solid black; padding: 2px;">1</span>	7531 ECA9 CC = 3

After the execution of the instruction the carry bit is removed from the sign bit in the contents of register. This bit is lost and the only method to determine whether a carry has occurred consists in

an analysis of the condition code after the execution of the instruction.

The ADD LOGICAL instructions allow addition of long numbers. An example is an algorithm for adding two double-precision numbers each of which occupies two fullwords, and the sum is also a double-precision number.

1. Add together the high-order halves of the numbers with the help of the instruction ADD (A or AR). The result is the high-order half of the sum.

2. Then, add together the low-order halves of the numbers with the aid of the instruction ADD LOGICAL (AL or ALR). The result is the low-order half of the sum.

3. If after the execution of the ADD LOGICAL instruction, the condition code is equal to 2 or 3, increment the high-order half of the sum by 1.

SUBTRACT LOGICAL 1F SLR RR  $R_1R_2$

$(R_1) - (R_2) \rightarrow R_1$

SUBTRACT LOGICAL 5F SL RX  $R_1X_2B_2D_2$

$(R_1) - (E_2) \rightarrow R_1$

In the execution of the SUBTRACT LOGICAL instruction, first is formed the complement of the second operand which is then added (on the ADD LOGICAL instruction) to the first operand. The condition code is set as follows:

- 1— non-zero difference; no carry from the sign bit occurs;
- 2— zero difference; a carry from the sign bit occurred;
- 3— non-zero difference; a carry from the sign bit occurred.

### Example

*Before instruction execution*

Register 2 1234 5678

Register 3 0123 4567

Instruction SLR 1F 3 2

1 0000 0000 0123 4567

— +  
1234 5678 EDCB A988

EDCB A988 EEEE EEFF

Instruction SLR 1F 2 3

1 0000 0000 1234 5678

— +  
1234 5678 FEDC BA99

FEDC BA99 1 1111 1111

*After instruction execution*

Register 3

EEEE EEFF CC = 1

Register 2

1111 1111 CC = 3

<i>Instruction</i> SLR 1F 2 2				
1	0000 0000	1234	5678	
—		+		
	1234 5678	EDCB A988		Register 2
				0000 0000 CC = 2
EDCB A988 <span style="border: 1px solid black; padding: 0 5px;">1</span>				0000 0000

These examples show that the condition code is set equal to:

- 1 if the first operand is less than the second operand,
- 2 if the operands are equal, and
- 3 if the first operand is greater than the second operand.

After the execution of the SUBTRACT LOGICAL instruction, the condition code is never set equal to 0.

The algorithm for subtracting double-precision numbers is similar to the above-described addition algorithm:

1. Subtract the high-order halves of the numbers with the aid of the instruction SUBTRACT (S or SR). The result is the high-order half of the difference.
2. Subtract the low-order halves of the numbers with the help of the instruction SUBTRACT LOGICAL (SL or SLR). The result is the low-order half of the difference.
3. If after the execution of the SUBTRACT LOGICAL instruction the condition code equals 1 (the low-order half of the minuend is less than the low-order half of the subtrahend), the high-order half of the difference is decremented by 1.

Here is a sample example of a program performing double-precision computations.

**Example.** Write a standard program to compute the sum of double-precision number array elements. Placed in linkage register 1 by the externally-stored program is the address of the table of parameters consisting of three words and having the following structure:

*Word*

- 1 — initial address of the array;
- 2 — address of the halfword containing the quantity of array elements in the form of an integer number  $n$ ;
- 3 — address of the first of the two subsequent words for writing the sum.

The program is given below.

000-003	STM	90 2 7 D 01C	Save the contents of registers 2 through 7
004-007	L	58 2 0 1 004	Address $n \rightarrow R_2$
008-00B	LH	48 3 0 2 000	$n \rightarrow R_3$ (load counter)
00C-00F	L	58 2 0 1 000	Array address $\rightarrow R_2$
010-011	SR	1B 4 4	
012-013	SR	1B 5 5	

014-017	LA	41 6 0 0 001	$1 \rightarrow R_6$
018-019	SR	1B 7 7	$0 \rightarrow$ index register
01A-01D	A	5A 4 7 2 000	Computation of the high-order half of the sum
01E-021	AL	5E 5 7 2 004	Computation of the low-order half of the sum
022-025	BC	47 C 0 F 028	Test for carry from the sign bit of the low-order half of the sum
026-027	AR	1A 4 6	$(R_4) + 1 \rightarrow R_4$
028-02B	LA	41 7 0 7 008	$(R_7) + 8 \rightarrow R_7$
02C-02F	BCT	46 3 0 F 01A	Test for loop termination
030-033	L	58 2 0 1 008	Sum address $\rightarrow R_2$
034-037	STM	90 4 5 2 000	Store sum
038-03B	LM	90 2 7 D 01C	Restore the contents of registers 2 through 7
03C-03D	BCR	07 F E	Exit from the subprogram

The program utilizes register 2 as the base register<sup>v</sup> for parameters. In our case parameters are referenced in sequence. Therefore, they must be assigned a base register, entering the required base value in it before use of each parameter. If the program needs the parameters concurrently, it is convenient to assign each of the parameters its own base register.

Register 3 is for the counter which controls the loop termination. The value of sum is accumulated in registers 4 and 5, while register 6 contains one—a constant necessary to add double-precision numbers. Register 7 is used as an index register in readdressing array elements.

Addition of double-precision numbers is carried out according to the above-described algorithm. The instruction BRANCH ON CONDITION (022) with a mask  $C_{16} = 1100_2$  bypasses incrementing the high-order half of the sum by 1, if no carry from the sign bit occurs in the execution of the instruction AL.

### Exercises

1. In the execution of this exercise arrange the source data and the field for the result in any locations of the main storage. Write the sequence of instructions which will perform the following actions:

- reverse the sign of a double-precision number;
- compare two double-precision numbers. After the execution of these instructions, the condition code should be equal to 0, if the numbers are equal, to 1, if the first number is less than the second number, and to 2, if the first number is greater than the second number;
- add a single-precision number occupying a fullword to a double-precision number;
- multiply a double-precision number by a single-precision number, the result being a triple-precision number;

(e) divide a triple-precision number by a single-precision number, the result being a double-precision quotient, and a single-precision remainder;

(f) multiply together two double-precision numbers, the result being a quadruple-precision number.

IMPORTANT! Do not forget that the number sign contains only the first word of those used for writing a multiple-precision number. Therefore, arithmetic operations cannot be performed on the second and subsequent words directly as the case is with conventional numbers. Think how this difficulty can be overcome in each of the exercises. Check also to see whether the execution of the written programs at various combinations of the operand signs is correct.

2. On the basis of Exercise 1, try to prepare a set of four subprograms for performing arithmetic operations on integer numbers of any precision.

The parameters of a subprogram are the address of operands and fields for the result in the main storage, and also the precision of the operands and result, i.e. the number of fullwords used for their record.

IMPORTANT! Probably, it will be difficult for you to write a subprogram at once in a general form. Therefore, first limit the precision of the numbers represented, i.e. consider the arithmetic operations, say, on numbers of triple precision or lower.

3. Write a standard subprogram for computation of  $n!$  for integer numbers not above 1000.

The subprogram parameters are:  $n$  in general register 0 and field address for the result in general register 1.

The results of the subprogram work are:  $n!$  in the assigned field of storage and the number of fullwords it occupies in general register 0.

## 4.9. Floating-Point Arithmetic Operations

The instructions performing operations on fixed-point numbers are best utilized for processing integer number data. In computations in which arithmetic operations are to be performed on any numbers, it is better to represent numbers data in the floating-point form. To carry out operations on floating-point numbers, the ES EVM instruction set includes 44 *floating-point instructions*. These instructions are executed with use of *floating-point registers* numbered 0, 2, 4 and 6, each being double-word long. The floating-point instructions considered in this section of the book utilize only these special-purpose registers, while all the other instructions with registers as operands make use of general registers.

The floating-point instructions may be of format RR or RX. The first operand in an instruction defines the floating-point register on

the content of which the operation will be carried out. The second operand in the RR format also defines a floating-point register, while the second operand in an instruction of the RX type specifies the address of a field in the main storage as usually.

It appears now worth while recalling the fact that floating-point numbers are represented with the aid of two elements: as a *fraction* or mantissa and an *exponent*. The exponent is a power of number 16 (the base of a numbering system used in the computer) the fraction is to be multiplied by to obtain the required number, i.e. the power to which 16 is raised indicates the number of positions the (hexa) decimal point should be shifted to give the true value of the number. For example, the number  $.C3A \times 16^2$  is really  $C3.A_{16}$ , while  $.10E \times 16^{-3}$  is  $.00010E_{16}$ . Therefore, the number with the mantissa  $m$  and exponent  $p$  has the value of  $m16^p$ .

In the machine storage the floating-point numbers are represented in the following format:

±	Characteristic	Fraction
Bit 0 1	7 8	31 or 63

The leftmost bit (bit 0) indicates the sign of the entire number, 0 for a positive number, 1 for a negative number. Note that this is different from binary integer arithmetic, in which the negative of a number is its 2's complement. The next 7 bits contain the *characteristic* which is numerically equal to the number exponent increased by 64 (the internal form of the exponent). This usage is called the *excess-64* notation. The characteristic becomes positive and there is no need to assign a separate digit place to the exponent sign. The remaining digital places are assigned to the mantissa. The mantissa can contain 6 or 14 hexadecimal digits, which corresponds to representing a *single-precision* or *double-precision* number. Single-precision (short) floating-point numbers are 32-bit full words. Double-precision floating-point numbers (for some reason called long) are double words.

The floating-point number representation in the ES EVM allows operations on numbers the absolute value of which is greater than  $16^{-64} \times 16^{-1} \approx 0.540 \times 10^{-78}$  and less than  $7F\text{ FFFFFFFF FFFFFFFF} = 16^{63} \times (1 - 16^{-14}) \approx 0.724 \times 10^{76}$ .

Any floating-point instruction may be in two versions performing operations on single-precision and double-precision numbers. The operands of the single-precision instructions should begin at a full-word boundary, while the operands of double-precision instructions, at a double-word boundary.

Single-precision numbers take half the storage space and arithmetic operations on them are carried out most often quicker than operations on double-precision numbers. However, the latter operations

essentially improve the computation precision. For the codes and mnemonics of the floating-point instructions, see Table 4.2.

Table 4.2

Instruction	RR Format		RX Format	
	Double precision	Single precision	Double precision	Single precision
LOAD	28 LDR	38 LER	68 LD	78 LE
*COMPARE	29 CDR	39 CER	68 CD	79 CE
*ADD	2A ADR	3A AER	6A AD	7A AE
*SUBTRACT	2B SDR	3B SER	6B SD	7B SE
MULTIPLY	2C MDR	3C MER	6C MD	7C ME
DIVIDE	2D DDR	3D DER	6D DD	7D DE
*ADD UNNORMALIZED	2E AWR	3E AUR	6E AW	7E AU
*SUBTRACT UNNORMALIZED	2F SWR	3F SUR	6F SW	7F SU
STORE			60 STD	70 STE
*LOAD POSITIVE	20 LPDR	30 LPER		
*LOAD NEGATIVE	21 LNDR	31 LNER		
*LOAD AND TEST	22 LTDR	32 LTER		
*LOAD COMPLEMENT	23 LCDR	33 LCER		
HALVE	24 HDR	34 HER		

The mnemonics of the floating-point instructions look like those of the fixed-point instructions, except for the special floating-point instruction HALVE. The mnemonics of the single-precision instructions include the letter E, and the double-precision instructions, the letter D (double). The mnemonics of the ADD UNNORMALIZED and SUBTRACT UNNORMALIZED single-precision instructions contain the letter U (unnormalized), and double-precision ones the letter W.

The asterisked instructions set the condition code of 0, 1 or 2 at a zero, negative, or positive result, respectively. The value of the condition code is set to 3 by these instructions when the result of the operation exceeds the maximum number that can be represented in the ES EVM in the floating-point form. The other instructions have no effect on the value of the condition code.

The double-precision instructions perform computations with 14 hexadecimal digits. In that, due to dropping significant digits in the alignment of the characteristics, the result precision may be reduced to 12 significant digits and less.

The single-precision instructions carry out computations with seven hexadecimal digits, but only six hexadecimal digits are saved in

recording the result in a register. The work of these instructions has no effect on the content of the right-hand half of the floating-point registers.

Certain instructions are described below.

### LOAD (LDR, LER, LD, LE)

The second operand is moved into the first operand. Recall once more the fact that in the single-precision instructions the first half of the register defined by the first operand does not change.

### COMPARE (CDR, CER, CD, CE)

These floating-point instructions compute the difference between the first and second operands and set the condition code depending on the result to 0, if the operands are equal, to 1, if the first operand is less than the second operand, and to 2, if the first operand is greater than the second operand.

### ADD (ADR, AER, AD, AE)

The second operand is added to the first operand and the sum is loaded in place of the first operand. The condition code is set equal to 0, if the result (sum) = 0, to 1, if it is less than 0, to 2, if the result is greater than 0, and to 3, if an exponent overflow has occurred.

Generally, the execution of the FLOATING-POINT instruction consists of the following steps:

- comparing the characteristics of the numbers;
- alignment of the characteristics;
- addition of the mantissas;
- sum normalization to the left or to the right.

Certainly, in particular cases, not all the above-mentioned actions may take place. No use may be made of such steps as alignment of characteristics, or normalization of the sum.

### Examples

#### 1. Adding the numbers

41 741020 98765401

41 612045 02100000

with the aid of the AE instruction produces the result

41 D53065 98765401

since the first half of the register defined by the first operand does not participate in the operation.

Adding the same numbers with the help of the AD instruction yields the result

41 D53065 9A865401

## 2. Addition of single-precision numbers

44 963100  
44 90261A

disturbs the normalization, since there occurs a carry from the most significant digit of the mantissa

+ 44 963100  
44 90261A  
441|26571A

and the result is

45 126571

Note, that after normalizing to the right, the seventh significant digit of the sum is lost.

## 3. In addition of single-precision numbers

45 100736  
C2 828304

it is necessary first to align the characteristics of the numbers. In this, the mantissa of the number having a smaller characteristic is shifted to the right until both numbers have similar characteristics. In our case, the mantissa of the second number should be shifted three hexadecimal digits to the right, and the result of addition will be

45 100736 0  
+  
C5 000828 3  
45 0FFF0D D

(Remember that in intermediate arithmetic operations on single-precision numbers saved are seven hexadecimal digits). The last step in the execution of the instruction will be normalizing the sum to the left, the final result being equal to 44 FFF0DD.

SUBTRACT (SDR, SER, SD, SE)

The second operand is subtracted from the first operand, and the result is substituted for the previous value of the first operand.

MULTIPLY (MDR, MER, MD, ME)

The operands are multiplied together and the result is moved into the place of the first operand. The multiplication of floating-point numbers uses only one register, unlike the multiplication of fixed-point numbers. The result is rounded off to 6 or 14 hexadecimal digits of the mantissa in compliance with the precision specified in the

instruction.

### DIVIDE (DDR, DER, DD, DE)

The first operand is divided by the second operand. Depending upon the precision specified in the instruction, 6 or 14 hexadecimal digits are computed in the mantissa of the quotient. The remainder is lost. The quotient is substituted for the previous value of the dividend.

### ADD UNNORMALIZED (AWR, AUR, AW, AU)

### SUBTRACT UNNORMALIZED (SWR, SUR, SW, SU)

All arithmetic floating-point instructions, except for the latter ones, produce the result in the normalized form, though they can work with unnormalized operands. In the execution of the ADD UNNORMALIZED and SUBTRACT UNNORMALIZED instructions, the result, if necessary, is normalized only to the right, and not to the left. Thus, for example, adding the numbers

44 963100

44 90261A

with the aid of the AU instruction, will produce the result

45 126571

i.e. it does not differ from the result produced by the use of the AE instruction.

However, adding the numbers

45 100736

C2 828304

with the help of the AU instruction will produce the result

45 0FFF0D

The ADD UNNORMALIZED and SUBTRACT UNNORMALIZED instructions are mainly used for precision control of the computations being performed. The only cause of precision loss in computations (in addition to rounding-off errors) is subtraction of numbers close in value (or addition of numbers close in absolute value, but with opposite signs). For example, in subtracting the numbers

44 123456

44 123421

the result equal to

40 350000

will have only two correct significant hexadecimal digits, though the initial numbers had six significant digits each. This is known as *significance exception*. In long computations the result may have no correct significant digits at all. In computations which may result

in an unpredictable significance exception, the program control of the computation precision should be organized. One of the methods may be parallel execution of actions on normalized and unnormalized numbers. Checking the number of high-order zero digits in the mantissa of an unnormalized result, one can define the precision of the computations in question. A mantissa of an unnormalized result equal to zero indicates a complete significance exception.

#### STORE (STD, STE)

A number from a register is stored in the memory. STD stores a double-precision (double-word) number and STE stores a single-precision (fullword) number.

#### LOAD POSITIVE (LPDR, LRER)

The absolute value of the second operand is moved into the place of the first operand. The condition code is set equal to 0 or 2, but not to 1, since the result cannot be negative.

#### LOAD NEGATIVE (LNDR, LNER)

The absolute value of the second operand, after its sign bit is changed, is moved into the place of the first operand. The condition code is set equal to 0 or 1.

#### LOAD AND TEST (LTDR, LTER)

This instruction works as the LOAD INSTRUCTION, but in addition it sets the condition code.

#### LOAD COMPLEMENT (LCDR, LCER)

The second operand with its sign bit changed is moved into the place of the first operand. The condition code may be set equal to 0, 1 or 2.

#### HALVE (HDR, HER)

This is a special floating-point instruction. The second operand is divided by 2 and moved into the place of the first operand. In fact, the mantissa of the second operand is shifted 1 bit to the right. The result of the operation is not normalized. If the second operand had a zero mantissa, the result would be zero.

To illustrate this, write a standard program of computation

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

The parameter  $x$  is a single-precision number. The result is to be obtained in the same form. The address of  $x$  is moved to linkage register 1. The result must be placed in floating-point register 0. The computations should be continued until the next in turn term of the series becomes less than  $10^{-6}$  in its absolute value.

For the flowchart of the algorithm, see Fig. 4.8

The algorithm is a single loop of the iteration type. Computed in the loop body is the consecutive term of the series (variable  $d$ ) which is then added to the sum of the preceding terms.

The program is given below.

000-003	LE	78 0 0 F 020	$1 \rightarrow y$
004-005	LER	38 2 0	$1 \rightarrow d$
006-007	LER	38 4 0	$1 \rightarrow n$
008-00B	ME	7C 2 0 1 000	$d \cdot x \rightarrow d$
00C-00D	DER	3D 2 4	$d : n \rightarrow d$
00E-00F	AER	3A 0 2	$y + d \rightarrow y$
010-013	AE	7A 4 0 F 020	$n + 1 \rightarrow n$
014-015	LPER	30 6 2	Obtain $ d $
016-019	CE	79 6 0 F 024	Compare $ d $ and $10^{-6}$
01A-01D	BC	47 2 0 F 008	Branch at $ d  > 10^{-6}$
01E-01F	BCR	07 F E	Exit from the subprogram
020-023		41 1000000	1
024-027		3C 10C6F7	$10^{-6}$

In connection with this program, it is worthy of considering a question: why are there no REGISTER SAVE and RESTORE instructions in this program? This question may be put by the thoughtful reader.

The matter is that the programming system does not provide the necessity of saving the contents of the floating-point in the program. Saving the contents of the required floating-point registers should be concerned for by the master program calling the subprogram. This convention is brought about by the different purposes of the general and floating-point registers.

The general registers are used in all programs and contain base values for various storage areas, indexes, etc. which are needed in the course of the execution of the whole of the program or some its segments.

The floating-point registers find their application only in the computation programs for storage of intermediate results of floating-point computations, for which reason saving the contents of floating-point

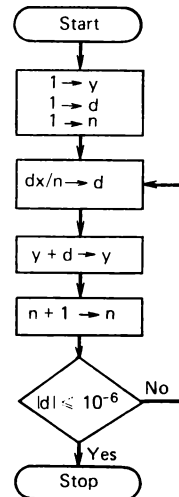


Fig. 4.8. Flowchart of  $y = e^x$  computation algorithm

registers should be handled by the program carrying out the computations.

As to the general registers, our program has no effect on any of them and, therefore, it contains no save and restore instructions at all.

### Exercises

1. Write a standard program for computing the cube root in the expression  $X = \sqrt[3]{N}$ . Make use of the Newton method, starting with a first guess  $X_0 = 1$  and computing the subsequent approximations by the formula

$$X_{k+1} = (2X_k + N/X_k^2)/3$$

Continue the computations until the difference between two successive estimates is less than  $10^{-6}$ . The subprogram parameter, the number  $N$  in the floating-point of the single-precision form, is moved into floating-point register 0. Place the result in floating-point register 2.

2. Write a standard subprogram for computing  $\sin x$ . Make use of sine expansion into a power series

$$\sin x = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \dots$$

Continue the computation (iteration) until a successive term is less than  $10^{-6}$ .

3. Using the identity  $\cos x = \sin(\pi/2 - x)$  alter the above exercise so that the subprogram is allowed to compute at its option either  $\sin x$ , or  $\cos x$ , depending on the reference.

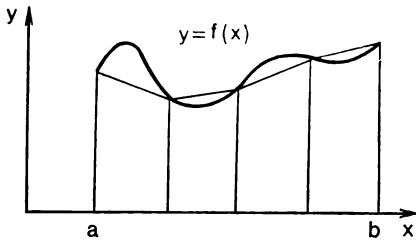


Fig. 4.9. Approximating the area under the curve by the trapezoidal rule

**IMPORTANT!** Build a subprogram with two entries. Reference to the first entry will cause computation of  $\sin x$ . The other entry carries out the conversion  $\pi/2 - x \rightarrow x$ , and then transfers control to the first entry.

SEE to it that general register 15 contains a correct base value for the subprogram regardless of what entry is referenced.

4. Write a standard subprogram for computing the value of a definite integral by the trapezoidal rule shown in Fig. 4.9. Divide the large curvilinear trapezoid into several narrow inscribed trapezoids, and then substitute a rectilinear trapezoid for each of them.

To approximate the area under the curve to the integral you may take the sum of the trapezoid areas; the approximation will be better with the narrower trapezoids.

In computations, the segment  $[a, b]$  under integration is generally divided into  $n$  equal parts. Then, the integral is evaluated by the formula

$$\int_a^b f(x) dx \approx \frac{b-a}{n} \left[ \sum_{h=0}^n f\left(a + \frac{h}{n}(b-a)\right) - \frac{f(a)+f(b)}{2} \right]$$

The subprogram parameters are integration limits  $a$  and  $b$ , number of division points  $n$ , and the address of the program for computing  $f(x)$ . The program for computing  $f(x)$  requires that  $x$  be in floating-point register 0, and the same register returns the computed value.

**IMPORTANT!** Do not forget to provide the program for computing  $f(x)$  with its own storage area and to execute properly words 2 and 3 of both storage areas used.

Generally, in computation of integrals, it is more convenient to prescribe the accuracy to which integral  $\varepsilon$  is to be evaluated, rather than division exponent  $n$ . Modify the program from the previous exercise accordingly. First specify  $n$  as a guess, say,  $n = 2$ , and compute the integral. Then, increase  $n$  twice and again evaluate the integral, and so on. Continue the doubling of the division exponent until the difference between two successive approximations is less than  $\varepsilon$ . Return the value obtained by the last computation to the calling program as the result.

#### 4.10. Operations on Data Codes

Almost all the above-considered instructions carry out operations on data of certain type, rather than on the contents of storage fields or registers. For example, the floating-point instructions handle solely floating-point numbers <sup>1</sup>.

However, many problems need operations which are carried out in a similar way for all types of data, regardless whether the data are numbers in any form of representation, text information, or even instructions. In fact, such operations are carried out on codes stored in storage fields or registers regardless of what data are represented by these codes.

The operations carried out on codes include movement and shift of data, comparison of codes, and Boolean (logical) operations.

**Data movement.** Many of the data movement instructions have been considered previously, for these instructions are mainly used

<sup>1</sup> The reader can take note of the fact that essentially the content of any four-byte field may be interpreted as a certain floating-point number, and therefore, the application of a floating-point instruction to such a field is legal. Make an attempt to conceive a problem in which a floating-point instruction would be utilized, say, for processing text information.

together with the fixed- and floating-point arithmetic operations. These are various load instructions performing *storage*  $\rightarrow$  *register* and *register*  $\rightarrow$  *register* movements and STORE INSTRUCTIONS carrying out *register*  $\rightarrow$  *storage* movements.

This section deals with the transfer instructions carrying out *storage*  $\rightarrow$  *storage* moves, and also instructions moving individual bytes.

Data transfers from one field in the main storage to another field in it are performed by the following instructions:

MOVE IMMEDIATE	MVI
MOVE CHARACTERS	MVC
MOVE NUMERICS	MVN
MOVE ZONES	MVZ
MOVE WITH OFFSET	MVO

The MVO instruction is not considered here, for it is used together with the decimal arithmetic instructions and will be fairly well described in the next section of this publication.

MOVE IMMEDIATE    92    MVI    SI     $I_2B_1D_1$   
 $I_2 \rightarrow E_1$

This instruction gives us an example of the SI format. The second operand in it is an immediate operand. One byte of data  $I_2$  contained in the instruction itself is transferred to the effective address  $E_1 = (B_1) + D_1$ .

The MVI instruction is convenient for us to transfer a data byte the contents of which is known in writing the program. This operation is executed far more rapidly than transferring a byte with the aid of the MVC instruction considered below, as in this event no separate reference to the storage for the second operand is needed.

MOVE CHARACTERS    D2    MVC    SS    L  $B_1D_1B_2D_2$

The MVC instruction moves a storage field starting at the initial address  $E_2 = (B_2) + D_2$  to the address  $E_1 = (B_1) + D_1$ . The number of bytes moved is given by the number  $L + 1$ . The maximum value which can be written in one byte is  $FF_{16} = 255_{10}$ . Therefore the MOVE CHARACTERS instruction can move from 1 (at  $L = 0$ ) to 256 consecutive bytes (at  $L = FF_{16}$ ). Data are moved in a series of bytes one after another, starting at the leftmost byte of the second operand field.

**Example.** The execution of the MOVE CHARACTERS instruction is illustrated by the following sample example.

*Before instruction execution*

Register 8	0000 6000
Storage 006000-07	1122 3344 5566 7788
Storage 006008-0F	90 A1 B2C3 D4E5 F607
Instruction MVC D2 05 8 001 8 009	

*After instruction execution*

Storage 006000-07	11A1 B2C3 D4E5 F688
Storage 006008-0F	Without changes

The MVC instruction is fairly simple, but if the fields defined by the first and second operands overlap, new abilities arise.

One of the possible applications of the MVC instruction consists in filling the whole of the field of the first operand with one character (the content of one byte). To execute such an instruction requires two instructions, one to move the required code to the first byte of the field, and the other to fill the other bytes of the field with this code.

**Example.** Fill 200 consecutive bytes with the hexadecimal value of FF starting with the byte having address 005000.

Suppose that base register 1 contains the value of 005000, then this operation can be executed by the following instructions:

```
MVI  92 FF  1 000
MVC D2  C6  1 001 1 000
```

The MVI instruction moves one byte of immediate data to the address of 005000. The MVC instruction propagates the value of the leftmost byte throughout the whole of the field. Each byte of the field, except for the last one, is moved to the nearest byte on the right. The data are moved in sequence, byte after byte, starting with the left-hand end of the field. Therefore, each byte first receives the required value of  $FF_{16}$  from the left, and then transfers it to the right.

199 bytes of the field (except for the first one) are filled with the aid of the MVC instruction, for which reason the length of the operands in the instruction should be specified equal to  $198_{10} = C6_{16}$ .

If the programmer needs to move data longer than 256 bytes, the fields may be divided into segments not in excess of 256 bytes, and the programmer may write several MVC instructions.

**Example.** Move 600 bytes to the address 004A00 starting at the address 004000. Suppose that base register 3 contains the value of 004000. Then, the move can be carried out by three MVC instructions:

```
MVC D2 FF 3 A00 3 000
MVC D2 FF 3 B00 3 100
MVC D2 57 3 C00 3 200
```

In a similar way all bytes of a long field can be filled with the same content. To this end, construct a chain of MVC instructions each of which carries out 'slide' movement.

The following two instructions move nibbles (halves of byte).

**MOVE NUMERICS    D1 MVN   SS   L B<sub>1</sub>D<sub>1</sub>B<sub>2</sub>D<sub>2</sub>**

This instruction moves each digit nibble, the low-order 4 bits of each byte, of the second operand to the corresponding position in the first operand area (field). The high-order nibbles (zones) remain unaffected. This instruction is fully similar to the MVC instruction, except for that the MVN instruction moves only the right-hand nibbles.

**MOVE ZONES        D3 MVZ   SS   L B<sub>1</sub>D<sub>1</sub>B<sub>2</sub>D<sub>2</sub>**

The MVZ instruction moves the zone portion, the first 4 bits of each operand byte from the second operand to the corresponding position in the first operand. The low-order nibbles (digits), the numeric portions of the bytes of the first operand, are left unchanged. This instruction is completely similar to the MVC instruction, except that only the left-hand nibbles are moved by the MVZ instruction.

The MVN and MVZ instructions are mainly used in various decimal number conversions, for which reason examples of their use are given below.

A data byte is stored and loaded into a general-purpose register by the following two instructions:

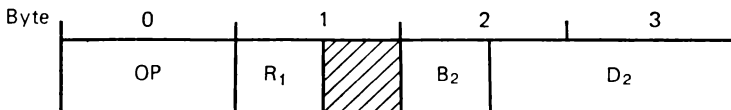
**STORE CHARACTER    42   STC   RX        R<sub>1</sub>X<sub>2</sub>B<sub>2</sub>D<sub>2</sub>**

It stores the contents of the last 8 bits of the general register R<sub>1</sub> at the address of the second operand.

**INSERT CHARACTER    43   IC   RX        R<sub>1</sub>X<sub>2</sub>B<sub>2</sub>D<sub>2</sub>**

This instruction loads the contents of the byte at the effective address E<sub>2</sub> into the last 8 bits of general-purpose register R<sub>1</sub>. The other bytes of register R<sub>1</sub> remain unchanged. Though the mnemonics of the IC instruction contain no word 'load', it can be fully referred to the instructions of this group.

**Data shift.** The term 'shift' is used to describe an operation of moving all bits of a general-purpose register one or several bit positions to the right or left. All the shift instructions are the RS type, but field R<sub>3</sub> is not used in them and may be filled with any information:



The shift instructions differ from one another in the shift direction (right-left), type (logical-arithmetic), and precision (single-double).

Therefore, there are eight shift instructions (Table 4.3).

The mnemonics of all the shift instructions are formed by the initial letters of the instruction designations (names):

SHIFT      RIGHT      —      LOGICAL,  
            LEFT      DOUBLE    ARITHMETIC

An example is the instruction SHIFT RIGHT DOUBLE ARITHMETIC — SRDA.

We shall not describe each of the eight shift instructions. Instead, we shall consider common characteristics of that group of instructions.

*Precision.* In the single-precision shift instructions use is made of one general-purpose register  $R_1$ , the content of which is shifted. In the double-precision shift instructions, the general-purpose register  $R_1$  must be of an even number. Shifted are the contents of two registers  $R_1$  and  $R_1 + 1$ , the contents being considered as one 64-bit code.

*Type.* In the logical shift instructions the sign bit does not differ from all the other bits for which reason shifted to the left or right are all 32 or 64 bits. Bits shifted out of a register are lost, and the vacated positions are replaced by zeros.

**Table 4.3**

Operation code	Shift direction	Type	Precision
88 SRL	Right	Logical	Single
89 SL	Left		
8A SRA	Right	Arithmetic	
8B SLA	Left		
8C SRDL	Right	Logical	Double
8D SLDL	Left		
8E SRDA	Right	Arithmetic	
8F SLDA	Left		

The arithmetic shift instructions cause the sign bit to remain where it is, with only the data bits being shifted, i.e. only 31 or 63 right-hand bits. In shifts to the left the vacated positions are replaced by zeros, while in the shifts to the right, the vacated positions are filled by the value of the sign bit, i.e. the sign bit is as if 'propagated' to the right.

If a code in the register is treated as a fixed-point number, then arithmetic shifting  $k$  positions to the left multiplies the number by  $2^k$ . Conversely, arithmetic shift  $k$  positions to the right is equivalent to dividing by  $2^k$ . Strictly speaking, the arithmetic shifts refer to the fixed-point arithmetic and should not be considered here. However, it would be unjustified to separate their description from the logical shift instructions.

*Condition code.* The arithmetic shifts set the condition code in the same way as do the other fixed-point arithmetic instructions. Thus, after execution of an arithmetic shift, the condition code is set to 0, 1 or 2, if the result is equal to zero, is less than zero, or is greater than zero, respectively. In an arithmetic shift to the left, the high-order significant bits of the number may be lost, if the value of the bit being lost differs from the value of the sign bit. If that is the case, a fixed-point overflow occurs, and the condition code becomes set to 3. The SHIFT LOGICAL instructions have no effect on the value of the condition code.

*Amount of shift.* The six rightmost bits of the effective address  $E_2 = (B_2) + D_2$  specify the amount of shift. The remaining 18 bits of the effective address  $E_2$  are not used. In the shift instructions the second address is accepted as a number of bits the shift is to be made, rather than a storage memory. By means of six bits we can specify maximum 63 and minimum 0. It is evident, that there is no point to shift more than 31 positions in the single-precision shifts.

The shifts are mainly used to isolate code portions stored in registers.

### Example

		Register 6	Register 7
<i>Before instruction execution:</i>		1234 89AB	0000 0000
<i>Instruction:</i>		<i>After instruction execution:</i>	
SRDL	8C 6 0 0 010	0000	1234 89AB 0000
SLA	8B 6 0 0 001	0000	2468 89AB 0000
SRA	8A 7 0 0 008	0000	2468 FF89 AB00
SRL	88 7 0 0 008	0000	2468 00FF 89AB
SLDL	8D 6 0 0 02C	F89A B000	0000 0000

**Character-oriented comparison instructions.** These instructions compare bit by bit the operands from left to right. If the operands are equal, the comparison terminates upon ending of the fields being compared. Otherwise, the execution of the operation is continued

until unmatched bits are encountered, and the operand containing a 1 in this bit is considered as greater. In all comparison instructions, the condition code is set equal to 0, if the operands are equal, to 1, if the first operand is less than the second operand, and equal to 2, if the first operand is greater than the second one.

There are four formats of the CHARACTER-ORIENTED COMPARISON instructions:

#### COMPARE LOGICAL      55 CL RX

This instruction compares the content of the register of the first operand with the content of the word defined by the second operand. As the case is with all word-oriented instructions, the address of the second operand must be aligned to the fullword boundary.

#### COMPARE LOGICAL      15 CLR RR

This is the same as the CL instruction, but the second operand is also in a general-purpose register. Those two instructions may be used together with the ADD LOGICAL and SUBTRACT LOGICAL instructions for organization of extended precision computations

#### COMPARE LOGICAL IMMEDIATE      95 CLI SI      $I_2B_1D_1$

This instruction compares contents of the byte stored at the address given as the first operand with the byte supplied as the immediate operand.

#### COMPARE LOGICAL CHARACTERS      95 CLC SS $LB_1D_1B_2D_2$

The CLC instruction compares the data in storage defined by the first operand with the stored data defined by the second operand. The comparison is made from left to right, bit by bit. The length of the data being compared is taken as equal to  $L + 1$  bytes, so that one CLC instruction can compare up to 256 bytes of data.

**Logical operations.** The Boolean or binary logic instructions refer to the bit manipulation operations, i.e. the value of any bit result is dependent only upon the values of the corresponding bits of the operands. The three logical operations of the ES EVM system are the AND operation, the OR operation, and the XOR (eXclusive OR) operation. Each of these groups consists of four instructions of different formats which perform similar actions and differ from one another only in the arrangement and length of the operands (see Table 4.4).

All the instructions of the Boolean logic have one feature in common. Their second operand is a *mask* determining in what way the bits of the first operand can be changed. The mask may be considered as a certain set of bits, each mask bit determining the action to be performed on the corresponding bit of the first operand.

Table 4.4

Instruction	RR	RX	SI	SS
AND	14 NR	54 N	94 NI	D4 NC
OR	16 OR	56 O	96 OI	D6 OC
EXclusive OR	17 XR	57 X	97 XI	D7 XC

In the Boolean logic instructions of the RR format, both the mask specified by the second operand and the source field specified by the first operand are in registers. Therefore, in the execution of the instruction, processed are exactly 32 bits (or 4 bytes), the length of a general-purpose register.

The instructions of the RX type use a fullword storage space as a mask to change the value of the bits contained in the general register. The mask specified by the second operand should be in the memory residing on a fullword boundary, while the first operand indicates the source field in the general-purpose register.

In the SI format instructions, one byte of immediate data specified by the second operand is a mask which specifies which of several logical decisions are to be made on the content of a byte from the memory.

In order to scan and to change the content of the source field found in the main storage, the Boolean logic instructions of the SS format use the mask occupying a storage area of the same length. The length of the source field specified by the first operand, and the length of the mask may be up to 256 bytes, respectively.

All Boolean logic instructions set the condition code. If the result equals 0, i.e. it consists solely of zero bits, the condition code is set equal to 0. If even one bit of the result is other than zero, the condition code is set to 1.

*The AND group instructions.* These instructions perform an operation of logical multiplication, bit by bit. The result replaces the previous value of the first operand and its bits are formed according to the rules given in Table 4.5.

Table 4.5

Mask bit	Corresponding bit in source field	Result bit in source field	Mask bit	Corresponding bit in source field	Result bit in source field
0	0	0	1	0	0
0	1	0	1	1	1

The AND operations may be used to set the specified bits of the source field to zero. The programmer should prepare a mask in which the corresponding bits are equal to zero. The one bits of the mask correspond to those bits of the source field which should remain in the previous state.

**Example.** Bits 0, 3, 4 and 6 of a certain byte in a subprogram are to be set to zero. The byte address is transferred to register 1. In order to set the above bits to zero, leaving the other bits unchanged, it is necessary to construct a mask in which bits 0, 3, 4 and 6 contain zeros, and the other bits contain ones. That mask has the form 65<sub>16</sub>:

	<div style="border: 1px solid black; padding: 5px; display: inline-block;">0 1 1 0   0 1 0 1</div>							
Bit	0	1	2	3	4	5	6	7

The required operation can be carried out by the instruction

NI 94 65 1 000

For one bits of the mask, the corresponding positions of the source field do not change, and where the mask contains a zero bit, its corresponding bit in the first operand will set to 0 or will remain in this state.

This example may seem a bit artificial, but the reader has not to invent an actual application for these instructions. It is enough to obtain a good understanding of significant and insignificant bits in the mask. Practically, instructions of Boolean logic can be encountered in any program containing logical processing of data, and a bit of practice will help you develop techniques of their use.

*The OR group instructions.* These instructions realize an operation of bit-manipulation logical addition. The result takes the place of the previous value of the first operand, and its bits are formed according to the rules shown in Table 4.6.

**Table 4.6**

Mask bit	Corresponding bit in source field	Result bit in source field	Mask bit	Corresponding bit in source field	Result bit in source field
0	0	0	1	0	1
0	1	1	1	1	1

Unlike the AND instructions, the OR instructions are used for setting the specified bits of the source field to 1. The programmer must prepare a mask in which the corresponding bits are equal to 1.

The zero bits of the mask correspond to those bits of the source field which are to remain unchanged.

*The eXclusive OR group instructions.* Like the other instructions of Boolean logic, the eXclusive OR instruction group consists of four instructions each of which carries out similar operations, but on different types of operands. Any variety of the eXclusive OR instruction performs the function of bit-by-bit modulo-2 addition, or, as it is called, a non-equivalence function. The result is defined by the following rules: two bits of equal value produce a zero in the result, while two different bits produce a resulting one. The results of how the mask bits act upon the source field bits are tabulated below (Table 4.7).

Tabla 4.7

Mask bit	Corresponding bit in source field	Result bit in source field	Mask bit	Corresponding bit in source field	Result bit in source field
0	0	0	1	0	1
0	1	1	1	1	0

The instructions of the EXCLUSIVE OR group may be used to *flip* the bits of the source field. Such an operation is known as *inverting* of bits.

Referring to the Table, a zero bit of the mask has no effect on the value of the corresponding bit of the source field, while its one bit inverts its corresponding bit in the source field.

The EXCLUSIVE OR group instructions find more practical applications than the other instructions of the Boolean logic. Some of those applications are exemplified below.

**Example.** Use of the EXCLUSIVE OR instructions of the RR and SS formats provides an easy method of clearing registers or storage fields. Thus, the execution of the instruction

XR 17 55

will result in entering a zero code in general register 5 regardless of its previous content. This is a best method of clearing the register, since the XR instruction is executed somewhat quicker than the SR instruction used before for the purpose.

As the result of the execution of the instruction

XC D7 63 0 700 0 700

a zero code will be entered in all bytes of the storage area starting at the address 000700,  $64_{16} = 100_{10}$  bytes in length.

**Example.** The fact that the EXCLUSIVE OR instruction allows data to be interchanged without need of additional storage may be somewhat unexpected, even for a person experienced in programming.

As you remember, to interchange  $a \leftrightarrow b$  before, one had to use an auxiliary work field  $r$  and carry out the following operations:

$$a \rightarrow r; \quad b \rightarrow a; \quad r \rightarrow b$$

The data  $a$  and  $b$  may be fields in the main storage having a significant length, and such an algorithm would require much of the main storage.

Now, let us study the following instructions:

XC D7 31 0 400 0 500

XC D7 31 0 500 0 400

XC D7 31 0 400 0 500

The reader should satisfy himself that the execution of these three instructions will result in an actual interchange of the 50-byte fields having initial addresses 000400 and 000500.

However, gaining storage, we lose time, as the XC instruction takes more time in execution than the MVC instruction moving a field of the same length.

The first of the two examples is based on the following property of the inversion operation: a bit inverted twice returns to its initial state.

Using the EXCLUSIVE OR instructions inverting certain bits of binary data, any information can be easily encoded, its decoding, without the key, being very difficult. The key in this event is represented by the mask of the EXCLUSIVE OR instruction. To decode the data, it is enough to apply to the data the same instruction that has been used in encoding.

If you find the above-described cipher too simple, it can be complicated in many ways. An example is using a key varying from byte to byte according to a certain rule.

We now consider an example of a program utilizing operations on codes.

Write a standard program of converting integer numbers from the fixed-point form to the floating-point form.

The subprogram parameter, an integer  $n$  is moved in linkage register 0. The result in the form of a floating-point double-precision number must be placed by the subprogram into floating-point register 0. The program is set forth below.

000-003	STM	90 2 3 D 01C	Save the contents of registers 2 and 3
004-005	LPR	10 2 0	$ n  \rightarrow R_2$
006-009	BC	47 7 0 F 014	Branch at $n \neq 0$
00A-00D	LD	68 0 0 F 038	

00E-011	LM	98 2 3 D 01C	Restore the contents of registers 2 and 3
012-013	BCR	07 F E	Exit from the subprogram
014-015	XR	17 3 3	$0 \rightarrow R_3$
016-019	SRDL	8C 2 0 0 008	Vacate the left-hand byte
01A-01D	STM	90 2 3 F 040	Store in the work field
01E-021	MVI	92 48 F 040	Store of characteristic
022-023	LTR	12 0 0	Test the sign
024-027	BC	47 A 0 F 02C	Branch at $n \geq 0$
028-02B	OI	96 80 F 040	Assign a minus sign
02C-02F	LD	68 0 0 F 040	
030-033	AD	6A 0 0 F 038	
034-037	BC	47 F 0 F 00E	To exit from the subprogram
038-03F		0000 . . . 0000	Constant: floating-point double precision 0
040-047			Double word for work field

The stages of conversion are shown in Fig. 4.10 by converting the number  $-365$ . Given below are explanations of certain instructions of the program.

1. Negative fixed-point numbers are represented in the machine in the complement form, while floating-point numbers are direct-code represented. Therefore, the absolute value of  $n$  has to be converted into the floating-point form with subsequent assignment of a required sign to the result. The instruction LPR (004) forms the absolute value of  $n$  in general register 2.

2. The instruction BC (006) tests the condition code for  $n \neq 0$ . If not so, a zero code is loaded in floating-point register 0, and an exit from the subprogram occurs.

3. The instruction XR (014) clears register 3, while the instruction SRDL (016) shifts the contents of registers 2 and 3 one byte to the right vacating the place for the characteristic and sign.

4. The instruction STM (01A) writes the prepared mantissa into work field 040-047.

5. The point in the formed mantissa is assumed to be after the eighth hexadecimal digit, since the source number is an integer, for which reason, the number characteristic should equal  $48_{16}$ . The characteristic is assigned by the instruction MVI (01E).

6. The three instructions LTR, BC and OI serve to assign the sign to the floating-point number formed, i.e. to set the zero bit to 1, if the source number is negative.

7. The instruction LD (02C) loads the number formed into floating-point register 0. However, this number can be unnormalized and the instruction AD (030) adds it to zero. After the floating-point arithmetic instructions, the result is produced in the normalized form.

In conclusion, let us dwell upon fairly popular application of the Boolean logic to SWITCHES.

In all the above examples of computation branching, the following actions were performed.

*Step 1.* Check on whether the condition is satisfied.

*Step 2.* Branch to one of the computation process branches.

The second step implemented by the BRANCH ON CONDITION instruction directly followed the first step. However, in certain

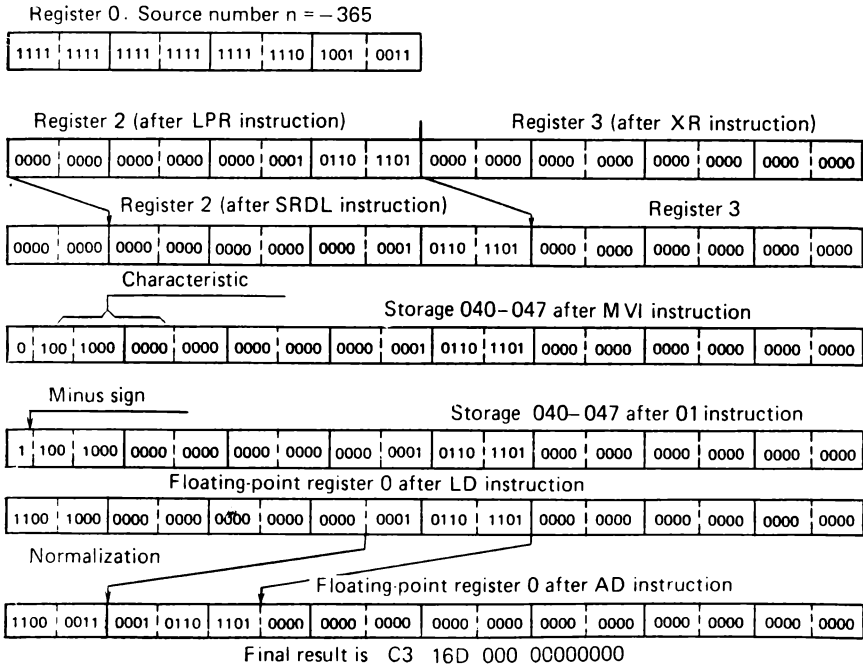


Fig. 4.10. Stages of converting the number  $-365$  into floating-point form

algorithms these steps are logically separated from each other. For example, problems are often encountered in which the condition is tested by one subprogram, while branching occurs in another subprogram. The check on condition may be complicated, requiring many data, and it will be no good to move it to the subprogram that will branch. In this event, the following procedure is used: the subprogram checking the condition sets a special condition code or *switch* defining further progress of computation, and transfers it to the subprogram being branched which checks the condition of the switch and chooses the required branch of the algorithm.

Another example is a subprogram carrying out various actions dictated by the requirements of the calling program. If these actions

are not so specific, that it is better to write a separate program for each of them, then it is good to transfer the switch to the subprogram as one of the parameters. The subprogram can, in turn, analyze this switch and carry out those operations which have been defined by its condition.

Generally, not strictly speaking, the switches serve as a 'notebook' for recording events capable of affecting the run of the computation process, if the response to an event in the program does not occur directly after the event.

In principle, use may be made of switches with many states. For instance, a switch represented in the memory by one byte can have up to 256 various states. In practice, however, two states are usually enough. A switch having only two states 'yes' or 'no' ('on'/'off') is called a bit switch. It is usually represented in the memory by one bit. The 1 state of the bit corresponds to the ON state of the switch, and the 0 state, to the OFF position.

A program may use several bit switches, and to save storage it is natural to compare with it various bits of one, or several bytes (if use is made of many switches). In this case, the NI (AND IMMEDIATE) instruction may be used to turn OFF several switches simultaneously, and the OI (OR IMMEDIATE), to turn them ON.

The use of the Boolean logic to test the status of the switches is somewhat inconvenient, as these instructions ruin the source field and precautions should be taken to save it. Therefore, the ES EVM set of instructions includes a special instruction which allows us to examine one or more bits within a byte in the memory.

TEST UNDER MASK    91   TM   SI     $I_2B_1D_1$

In the execution of the TM instruction the mask specified by the operand  $I_2$  is compared with one byte of the memory, at the address of the first operand. The mask is formed by the programmer using the following rules.

1. If a bit in the byte of data is to be tested, its corresponding bit in the byte of the mask should be set to 1.
2. If a bit in the byte of data is not to be tested, then its corresponding bit in the byte of mask should be set to 0.

The TM tests only those bits of the byte of data which have corresponding 1 bits in the mask. In the execution of this instruction the contents of the byte under testing do not change. The TM instruction sets the condition code to indicate the result of the test as follows:

CC = 0: 1. All tested bits are 0, or

2. Mask is 0.

CC = 1: Tested bits are 'mixed'; some are 0 and some are 1.

CC = 3: Tested bits are all 1.

After the execution of a TM instruction, the condition code is never set to 2.

The TM instruction allows the programmer to test any bit switch, or simultaneously several switches in one byte to see whether it (they) is (are) in the ON/OFF position, with no effect on their state.

Of the other possible uses of the TM instruction, we may indicate here determining the sign of a number situated in the memory by testing the sign bit.

### Exercises

1. In many problems, say in simulating various processes, use is made of random numbers. To obtain sequences of random numbers various generators have been developed. One of the first generators was proposed by the American scientist, John von Neumann, who utilized the mid square technique. Let  $X_n$  be the last random number obtained, then the number  $X_{n+1}$  is obtained by squaring  $X_n$  which will give us a double-precision number. Next, take the middle 32 digits of the  $X_n^2$  as the random number  $X_{n+1}$ .

Write a standard subprogram as a random-number generator after von Neumann. The parameter of the subprogram—the previous random number  $X_n$ —is transferred to general-purpose register 1. The result, a number  $X_{n+1}$ , should be placed in the same register.

2. General-purpose register 0 contains a certain code. Prepare a program which will reverse the bits in it, i.e. the leftmost bit will become the rightmost, bit 1 will take the place of bit 30 and so on, bit 31 will be moved to the place of bit 0.

3. Write a standard subprogram for computing the functions of Entire ( $X$ ), an integral part  $X$ . The Entire ( $\bar{X}$ ) is a maximum integral number which does not exceed  $X$ . For example,

$$\begin{aligned}\text{Entire}(3.5) &= 3, \text{Entire}(2) = 2, \text{Entire}(-3.6) = -4, \\ \text{Entire}(-2) &= -2\end{aligned}$$

The subprogram parameter—the  $X$  floating-point double-precision number — is transferred to floating-point register 0. The result in the same form is to be placed in the same register.

4. Given: the function of Sign of ( $X$ ). The sign of  $X$  is determined by the formula

$$\text{Sign}(X) = \begin{cases} -1 & X < 0 \\ 0 & X = 0 \\ 1 & X > 0 \end{cases}$$

Write a standard subprogram for computing Sign ( $X$ ). The subprogram parameter, an integral number  $X$ , is transferred in register 1. Place the result in general-purpose register 0. Could you write a program consisting of five instructions?

### 4.11. Decimal Arithmetic Instructions

The modern computers are most suitable for performing arithmetic operations in binary notation. The fixed- and floating-point binary arithmetic is represented by various instructions indispensable to problems containing much computation work. However, computations are far from being the only field of computer applications. Use of computers in the solution of various economic problems is no less important and becomes nowadays most popular. In contrast to scientific and engineering calculations, in solving economic problems, one has to handle very much input and output data frequently amounting to hundreds of thousands and millions of numbers.

Another specific feature of the economic problems is that they utilize computers mainly for various conversions, comparison and collating of data, rather than for computations which are rare. Because of this, it is often not paying to convert numerical values into binary notation. The computing practice has shown that it is better to store economic information in the form, it comes to the computer from a man (and will be utilized by the man after processing in the computer), i.e. in the decimal representation of numbers.

To facilitate the work with decimal numbers, the ES EVM set of instructions includes eight instructions of decimal arithmetic. Six of those instructions are dealt with in this section, and the other two, in Sect. 4.13.

All instructions of the decimal arithmetic are in the SS format, but the operands may vary in length. Therefore, each operand in the instructions has its own indicator of length

Byte            0            1            2            3            4            5

OP	L <sub>1</sub>	L <sub>2</sub>	B <sub>1</sub>	D <sub>1</sub>	B <sub>2</sub>	D <sub>2</sub>
----	----------------	----------------	----------------	----------------	----------------	----------------

The length indicators may range from 0 to 15 bytes. The decimal arithmetic instructions do not require storage alignment, nor do they use general registers (except for the base registers). In order to store intermediate results of computations, the programmer should assign work areas in the memory, 1 to 16 bytes long, depending upon the digit capacity of the numbers used.

The decimal arithmetic instructions perform operations on *packed decimal* operands. Each byte of the field allocated to store a number contains two decimal digits. An exception is the rightmost byte the four low-order positions of which must contain the number sign. To represent the sign, use is made of hexadecimal digits A, B, C, D, E and F. A number containing digits A, C, E, or F in the sign position is treated as positive. To designate a negative number, its sign nibble must contain B or D.

Several examples of representing decimal numbers in storage are as follows:

3	7	2	9	1	C	+37291		
0	0	0	0	0	2	8	F	+28
					5	D	-5	

Though the signs of the operands of decimal arithmetic instructions can be represented by several methods, the sign of the result is strictly defined. A positive result of executing any decimal arithmetic instruction always contains the hexadecimal digit C in the sign position. A negative result contains D in this position.

#### ADD PACKED DECIMALS    FA   AP   SS   L<sub>1</sub>L<sub>2</sub>B<sub>1</sub>D<sub>1</sub>B<sub>2</sub>D<sub>2</sub>

The second operand is added to the first one, and the sum is placed in the first operand location. The result sign is defined in compliance with the algebraic rules and placed in the rightmost hexadecimal position of the result. A zero result always has a plus sign. The condition code is set to 0, 1 or 2, if the result is equal to zero, less than zero, or greater than zero, respectively. The result never goes beyond the field specified by the first operand. If the result (sum) is too big to fit the field specified for it, a *decimal overflow* occurs, the condition code is set to 3, and the high-order values of the sum digit are lost.

#### Example

Before the instruction execution, the fields of the operands are as follows:

Storage 000973-75    01874C

Storage 000460-61    267C

*The instruction to be executed:* AP   FA 2 1 0 973 0 460

*After instruction execution:*

Storage 000973-75    02141C

Storage 000460-61    w/o changes

The execution of any decimal arithmetic instruction has no effect on the second operand field. Therefore, all the sample examples that follow, contain only the content of the first operand field after the execution of the instruction.

**Example***Before instruction execution:*

Register 4            0000 8720

Storage 008720-23    0372 136C

*The instruction to be executed:* AP FA 3 1 4 000 4 002*After instruction execution:*

Storage 008720-23    0372 272C

This example shows that the operand fields in the decimal arithmetic instructions may overlap. If that is the case, their right-hand bytes must coincide.

In the execution of any instruction of decimal arithmetic, checks are made on whether the operand fields contain packed decimal numbers. All nibbles of an operand field, except for the sign nibble, must contain the hexadecimal digits from 0 to 9, while the sign nibble should have values ranging from A to F. If the operand fields contain an illegal code of a digit or a sign, the execution of the instruction is interrupted, and the program containing such an instruction cannot be continued.

**Example***Before instruction execution:*

Register 6            0000 4400

Storage 004400-01    392C

Storage 004420-23    0000 607C

*The instruction to be executed:* AP FA 1 3 6 000 6 020*After instruction execution:*

Storage 004400-01    999C

In this example, no decimal overflow occurs. Though the length of the second operand is greater than the length of the first operand, the excessive bytes of the second operand field contain 0's, and the field of the first operand is large enough to hold the sum.

Generally, to use such instructions is not recommended, since the programmer should know in this case for certain that no overflow will occur in adding the quantities encountered in the program.

**SUBTRACT PACKED DECIMALS      FB SP SS L<sub>1</sub>L<sub>2</sub>B<sub>1</sub>D<sub>1</sub>B<sub>2</sub>D<sub>2</sub>**

This instruction subtracts the contents of the second operand from the contents of the first operand. The result is placed in the first operand location. The condition code is set in the same manner as in the instruction ADD PACKED DECIMALS. This instruction is similar to the AP instruction. The only difference is that the machine automatically reverses the sign of the second operand during the execution of the instruction.

**Example***Before instruction execution:*

Register 10	0000 6E00
Storage 006E00-02	0451 8C
Storage 006E50-52	0084 4C

*The instruction to be executed:* SP FB 2 2 A 000 A 050*After instruction execution:*

Storage 006E00-02	0367 4C
-------------------	---------

This example shows a simple case of subtraction. Really,  
 + 4518 minus + 844 equals + 3674.

**Example.** Under the conditions of the above example, after the execution of an instruction SP FB 2 2 A 050 A 000 the field of the first operand will take the form

Storage 006E50-02	0367 4D
-------------------	---------

In the latter example subtracting a bigger number from a smaller one produces a negative difference.

COMPARE PACKED DECIMALS F9 CP SS L<sub>1</sub>L<sub>2</sub>B<sub>1</sub>D<sub>1</sub>B<sub>2</sub>D<sub>2</sub>

This instruction is similar to the instruction SUBTRACT PACKED DECIMALS. The only difference is that the result is not saved and the instruction is used solely to define the condition code. Both operands remain unchanged. Like in the other instructions of comparison, the condition code is set equal to 0, if the operands are equal to one another; equal to 1, if the first operand is less than the second operand; and equal to 2, if the first operand is greater than the second operand. The result is not stored, and therefore, no overflow can occur.

Note, that the CP instruction compares namely decimal numbers defined by the operands, rather than their representations in storage. Thus, the application of the CP instruction to the fields

1562 9C
0000 1562 9A

will set the condition code to 0, since both fields represent the same number + 15629.

Any legal representations of the plus sign (A, C, E, F) are considered as equal. The characters representing the minus sign (B and D) are considered as equal too. The packed decimal magnitude + 0 is considered to be equal to the packed decimal magnitude - 0.

ZERO AND ADD PACKED DECIMALS F8 ZAP SSL<sub>1</sub>L<sub>2</sub>B<sub>1</sub>D<sub>1</sub>B<sub>2</sub>D<sub>2</sub>

This instruction moves a packed decimal number from the field defined by the second operand to the first operand location. If the

length of the first operand field is greater than the length of the second operand field, then the unfilled high-order positions of the first operand are packed with zeros. If the length of the first operand field is less than that of the second operand field, the high-order digits of the result are lost. If lost are non-zero digits, a decimal overflow occurs, and the condition code is set to 3.

When no overflow occurs, the condition code is set to 0, if the number being moved equals 0. It is set to 1, if the number is less than zero, and equal to 2, if the number is greater than zero.

The ZAP instruction differs from the other instructions of decimal arithmetic in that only the second operand is tested to see whether its digit and sign codes are correct. Therefore, the right-hand bytes of the operand may not coincide, if the operand fields overlap. It is only necessary to see that the rightmost byte of the first operand field is to the right of the rightmost byte of the second operand field (or is aligned with it).

In fact the ZAP instruction is executed exactly as the AP instruction, but the field of the first operand is considered to contain a decimal 0.

The ZAP instruction finds two major applications shown below by examples.

### Example

*Before instruction execution:*

Register 1	0001 5670
Storage 015670-75	A1B2 C3D4 E5F6
Storage 000600-00	0 C

*The instruction to be executed:* ZAP F8 5 0 1 000 0 600

*After instruction execution:*

Storage 015670-75	0000 0000 000C
-------------------	----------------

This example shows a simple method of clearing any field to decimal zero. Note, that to use an SP instruction for the purpose is not safe, since you must be certain that before the execution of the instruction, the field being cleared contained a valid packed decimal number, otherwise, the execution of a program containing such an SP instruction will be interrupted because of invalid data.

### Example

*Before instruction execution:*

Register 1	0000 5200
Storage 005200-07	any content
Storage 005400-02	7124 9D

*The instruction to be executed:* ZAP F8 7 2 1 000 1 200

*After instruction execution:*

Storage 005200-07	0000 0000 0071 249D
-------------------	---------------------

This example shows the use of the ZAP instruction for extending the field of the second operand. Such an extension may be needed before MULTIPLY and DIVIDE instructions.

MULTIPLY PACKED DECIMALS FC MP SS L<sub>1</sub>L<sub>2</sub>B<sub>1</sub>D<sub>1</sub>B<sub>2</sub>D<sub>2</sub>

This instruction multiplies the second operand (multiplier) by the first operand (multiplicand). The result of the multiplication is placed in the field of the first operand. The sign of the product follows the rules of algebra, though one or both operands are equal to zero. Thus, +126 multiplied by -0 equals -0.

The length of the multiplier should not exceed eight bytes and should be less than the length of the multiplicand. No overflow is permitted in the work of the MP instruction. To this end, the field of the multiplicand must have the number of high-order zero bytes equal to the number of digits plus sign in the multiplier (the size of the second operand field), otherwise the computer stops processing because of invalid data.

The condition code remains unchanged.

**Example.** Multiply together the packed decimals contained in the fields

Storage 024600-02	1837 5C
Storage 024930-31	649D

The result must be placed in the field five bytes long, at address 024610.

Let general-purpose register 2 contain the value of base 024600. In this case, the required operation can be accomplished by two instructions:

```
ZAP F8 4 2 2 010 2 000
MP FC 4 1 2 010 2 330
```

The first instruction locates the multiplicand in the low-order bytes of the field allocated for writing the product and clears its high-order bytes.

The result of its execution will be as follows:

Storage 024610-14	Multiplicand
	<u>18375C</u>

0000

The high-order zero places the total length of which equals the length of the multiplier field

After the execution of the ZAP instruction, the result field contains two high-order zero bytes, which is needed for the subsequent mul-

tiplication, since the length of the multiplier equals two bytes. After the execution of the MP instruction, the result field will contain the product

Storage	024610-14	0119 2537 5D
DIVIDE PACKED DECIMALS		FD DP SS L <sub>1</sub> L <sub>2</sub> B <sub>1</sub> D <sub>1</sub> B <sub>2</sub> D <sub>2</sub>

In contrast to the other arithmetic instructions, the result obtained in the execution of the DP instruction is unique. We must consider both the quotient and the remainder. This instruction divides the contents of the first operand (the dividend) by the contents of the second operand (the divisor). The quotient and the remainder are placed in the field of the first operand. The dividend, divisor, quotient, and remainder are considered as integer numbers. The remainder is placed in the low-order bytes of the first operand field, and its length in bytes is equal to the length of the divisor.

To write the quotient, use is made of the remaining L<sub>1</sub>—L<sub>2</sub> high-order bytes.

The sign of the quotient follows the rules of algebra, depending on the signs of the dividend and divisor. The remainder always takes the sign of the dividend, though it may be equal to zero.

The condition code remains unchanged.

In the course of work with the DP instruction, the following rules should be observed.

1. The length of the divisor should not exceed eight bytes and must be less than the length of the dividend.
2. Both fields must contain valid packed signed decimal numbers.
3. The field of the dividend must contain as many zero bytes, as the length of the divisor is.

The first two requirements apply to all cases of decimal division. A failure to follow these rules will inevitably interrupt the program containing the erroneous instruction. The last requirement is less strict. Its observance guarantees that the quotient fits the field allocated to it. The programmer who is sure that a less number of bytes will be enough to write in the quotient may respectively assign a smaller field for the dividend, though it is not recommended.

**Example.** The fields and work areas utilized in this example have the following contents:

Dividend	Storage	008200-04	0576 4905 4C
Divisor	Storage	008205-06	418C
Field for the result	Storage	008207-0D	0000 0000 0000 0C

Assume that the result field contains a decimal zero, though its content is of no importance. It is assumed that register 2 contains the value of base 008000.

The first action is to enter the dividend into the result area, for which purpose use is made of the instruction

ZAP F8 6 4 2 207 2 200

After the execution of the ZAP instruction, the field of the result will contain

Storage 008207-0D    0000 0576 49054C  
Dividend

Now we may carry out the division on the instruction

DP FD 6 1 2 207 2 205

after which the field of the result will take the form

Storage 008207-0D    0001 37916C 166C  
Quotient    Remainder

Note that in this example the length of the result field is chosen equal to the sum of the field lengths of the dividend and divisor. Therefore, the length of the quotient occupies the left-hand five bytes of the result field. The remainder, two bytes in length, is situated at the right-hand end of the result field.

The above-mentioned instructions prove the fact that

$$57649054 = 137916 \times 418 + 166$$

MOVE WITH OFFSET F1 MVO SS L<sub>1</sub>L<sub>2</sub>B<sub>1</sub>D<sub>1</sub>B<sub>2</sub>D<sub>2</sub>

The operation carried out by this instruction is not arithmetic, but it is used together with the decimal arithmetic instructions. Therefore, it is advisable to describe the MVO instruction in this section of the book.

The sign nibble (the four low-order bits of the right-hand byte) of the first operand remains unchanged, while the second operand is shifted a nibble to the left and placed close to the sign position of the first operand. If the operation result is greater than the field of the first operand, high-order (left-hand) digits of the second operand are lost. If the second operand is less than the first operand, zeros are filled in the vacated positions.

The MVO instruction is related to the operations performed on codes and there is no checking for valid decimal (packed) data.

The fields are processed byte by byte from right to left. Therefore, the operand fields may overlap, but so that the rightmost byte of the first operand field is to the right of the rightmost byte of the second operand field.

The condition code is not affected by the MVO instruction.

**Example.** Before instruction execution:

Register 5            0000 6000  
Storage 006100-04    9123 4567 89  
Storage 006105-07    1111 1C

The instruction to be executed: MVO F1 2 4 5 105 5 100

After instruction execution:

Storage 006105-07    5678 9C

**Example.** Under the conditions of the previous example, after the execution of the instruction

MVO F1 4 2 5 100 5 105

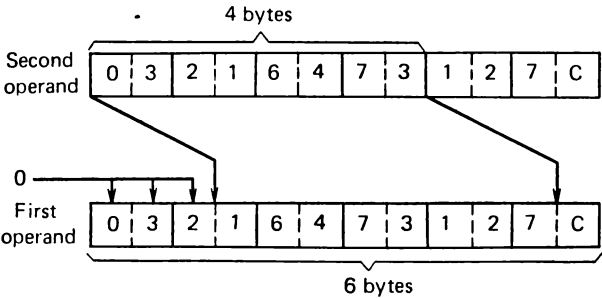
the result will be as follows:

Storage 006100-04    0001 1111 C9

**Example.** Before instruction execution:

Register 3            0000 7000  
Storage 007400-05    0321 6473 127C

The instruction to be executed: MVO F1 5 3 3 400 3 400  
We show the execution of this instruction in more detail:



The result of the operation:

Storage 007400-05    000 3216 473C

In this example the MVO instruction has moved the number three positions to the right which is equivalent to dividing by 1000. The sign bit is not affected by the operation.

We shall consider below an *assumed decimal point*. In the execution of the decimal arithmetic instructions the operands are considered to be integer numbers. However, problems often are encountered in which data contain digits after the decimal point. The machine does not provide facilities for indicating explicitly the position of

the decimal point in a number, and therefore, data *scaling* must be resorted to in performing arithmetic operations on fractional decimal numbers.

The scaling consists in multiplying the number by the appropriate power of 10 with a view to handling an integer number in the subsequent operations. Thus, instead of handling the number 214.86, we may deal with the number 21486, remembering that it is 100 times the actual value.

The scaling can be explained in another way. We represent data in the memory in the form of integer numbers, assuming that they contain the decimal point.

2	1	4	8	6	C
---	---	---	---	---	---



The assumed position of  
the decimal point

When dealing with scaled numbers, one has to take into account the position of the assumed decimal point. Shown below are certain techniques of carrying out arithmetic operations on fractional decimal numbers. The position of the assumed decimal point is shown in the figures by an arrow.

The execution of arithmetic operations on numbers with the assumed decimal point is based on the fact that such numbers can be represented in the memory by several methods. For example, the number 214.86 can be represented not only as  $214.86 = 21486 \times 10^{-2}$ , but as  $214860 \times 10^{-3}$ ,  $2148600 \times 10^{-4}$ , etc. In the memory, this corresponds to the following:

0	2	1	4	8	6	0	C
---	---	---	---	---	---	---	---

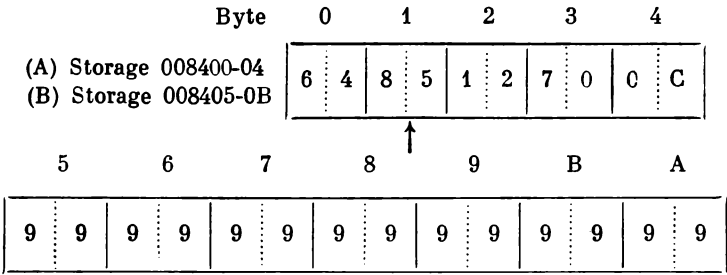
2	1	4	8	6	0	0	C
---	---	---	---	---	---	---	---



Representing the number in the memory by one of the possible methods, we may obtain the required number of decimal digits after the decimal point. The principal auxiliary operation in performing arithmetic operations on decimal fractional numbers is a change in the position of assumed decimal point, i.e. transition from one number representation to another. Additional places after the assumed decimal point can be included in or removed from a packed decimal number by multiplying or dividing by 10, 100, 1000, etc. However,

the decimal multiplication and division instructions are executed rather slowly, and we shall describe here an additional method of changing the position of the decimal point with the aid of MOVE instructions.

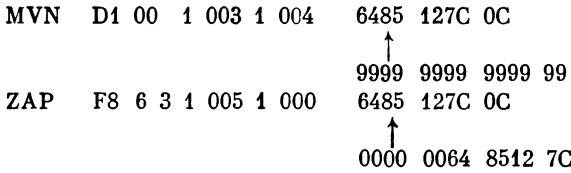
In the examples that follow, use is made of the same fields which contain the following data before the execution:



The A field contains the packed decimal number 648.5127 with six digits following the assumed decimal point. The content of the B field is not used in the examples and is given only for illustration. Of course, the B field may contain any other information.

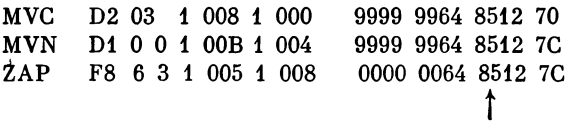
It is assumed that general-purpose register 1 contains the value of base 008400.

**Example.** Move the content of field A into field B by shifting the position of the assumed decimal point two positions to the right. If the content of field A will not be used in future, this operation can be carried out with the aid of two instructions:



The contents of fields A and B are shown in the first column after the execution of each instruction. The MVN instruction moves the number sign into the byte in which it must be after the shift. The ZAP instruction moves the left-hand four bytes of field A into field B after clearing the latter.

If the content of field A may not be changed, then carrying out the same operation will take three instructions. Only the content of field B after the execution of the instruction will be shown herein:



The MVC instruction moves the first four bytes of field A to field B (starting with the fourth byte of field B). In this case, the last digit and number sign are not moved in field B. The first three bytes of field B are not changed. The MVN instruction moves the number sign to the last byte of field B, and the subsequent ZAP instruction clears the first three bytes.

The similar instructions may be used in shifting the decimal point any even number of positions to the right.

**Example.** Move the content of field A to field B, having shifted the assumed decimal point three positions to the right. To carry out this operation, we have to use the MVO instruction:

```

MVO  F1 6 2 1 005 1 000    0000 0006 4851 29
MVN  D1 00 1 00B 1 004    0000 0006 4851 2C
                                     ↑

```

On the MVO instruction three bytes of field A (648512) are placed into field B to the left of its low-order nibble. Since the number of the bytes moved is less than the length of the receiving field, the digits being moved are supplemented with filling zeros on the left. Like in the previous example, the MVN instruction is used to move the number sign.

The MVO and MVN instructions are used to move the decimal point any odd number of positions to the right.

Note that in this example only three digits follow the assumed decimal point, and the last significant digit of the number is lost. In this case often the result needs to be rounded.

The rounding operation on a positive number involves increasing a digit by one if the digit immediately to the right is 5 or greater, and leaving it unchanged if the digit to its right is less than 5. In our example, to perform rounding we must add the decimal constant + 500 which we shall locate at the address 00840C:

Storage 00840C-0D

5	0	0	C
---	---	---	---

The move of the number 648.512700 from field A to field B with rounding to three digits after the decimal point, is carried out by the following instructions:

```

ZAP  F8 6 4 1 005 1 000    0000 6485 1270 0C
                                     ↑
AP   FA 6 1 1 005 1 00C    0000 6485 1320 0C
                                     ↑
MVO  F1 6 4 1 005 1 005    0000 0006 4851 3C
                                     ↑

```

The ZAP instruction moves the number from field A to field B, since rounding in field A will 'spoil' the source number. The next AP instruction for rounding to three digits after the decimal point adds the number 500 to the content of field B. After the execution of the MVO instruction, the decimal point is placed before the third digit on the right, and the other places of the number are truncated.

**Example.** Move the content of field A to field B by shifting the assumed decimal point one place to the left.

This operation again needs the MVO instruction:

MVO	F1 6	4 1 005 1 000 0006	4851 2700 C9
MVN	D1 00	1 00B 1 004 0006	4851 2700 CC
NI	94 0F	1 00B 0006	4851 2700 0C

↑

The MVO instruction moves the content of field A to field B by shifting one place to the left. The MVN instruction moves the number sign to the low-order nibble not affected by the previous instruction. The NI instruction is used to clear the last digital position of the number.

Pay attention to the mask of the NI instruction. It is composed so that all bits of the high-order nibble of the first operand are set to 0 while the bits of the low-order nibble containing the sign remain unchanged.

**Example.** Move the content of field A to field B by shifting the position of the assumed decimal point four places to the left.

MVC	D2 04 1 005 1 000	6485 1270 0C99 99
XC	D7 01 1 00A 1 00A	6485 1270 0C00 00
NI	94 F0 1 009	6485 1270 0000 00
MVN	D1 00 1 00B 1 004	6485 1270 0000 0C

↑

In this example note the use of the XC instruction to clear the last two bytes of field B.

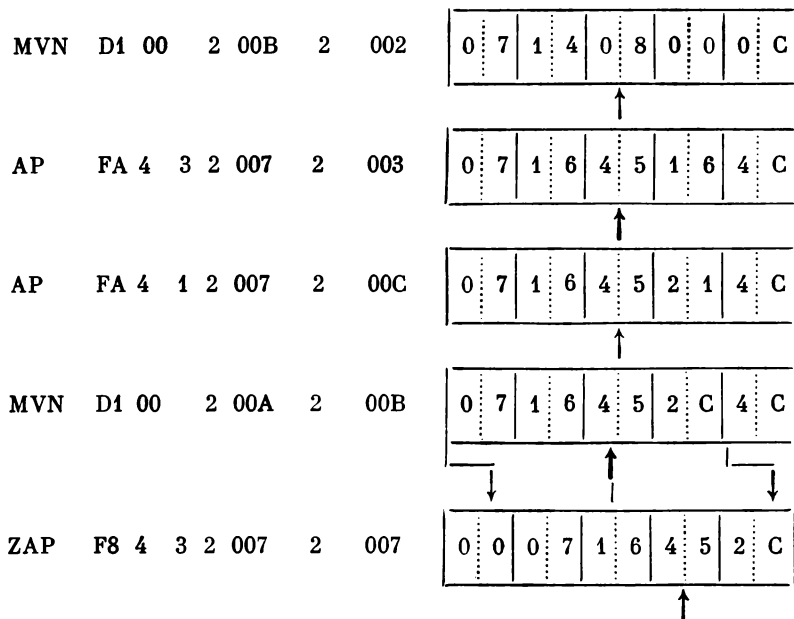
The last two examples illustrate the techniques of shifting the position of an assumed decimal point to the left odd and even numbers of positions, respectively.

Let this section be completed by describing the rules for adding two fractional decimal numbers.

1. If both fractional numbers contain an equal number of digits following the assumed decimal point, they can be added together with the aid of an AP instruction, like integer numbers.

2. If the numbers contain different number of digits after the decimal point, align the positions of the points.





The execution of these instructions results in the following computations  
 $7140.8 + 23.7164 = 7164.5164 \approx 7164.52$

### Exercises

1. Prepare a set of standard subprograms to perform arithmetic operations on fractional decimal packed numbers. The parameters of subprogram are the addresses of the operands and field for the result, lengths of the fields and position of the assumed decimal point in them. Think over a format for the transfer of parameters most suitable for the user.

2. Prepare a set of subprograms for carrying out arithmetic operations on decimal numbers of whatever length. Use a double word containing 15 digits and a sign as an elementary item. The decimal multiple precision arithmetic is less complicated than the binary arithmetic, because each item in it contains a number sign.

3. The mathematician William Shanks had spent 20 years of his life to give pi ( $\pi$ ) to 707 decimal places.

IMPORTANT! Make use of the decimal multiple precision arithmetic for the purpose (see Exercise 2).

To calculate  $\pi$ , use may be made of Machin's formula:

$$\pi = 16 \left( \frac{1}{1 \cdot 5} - \frac{1}{3 \cdot 5^3} + \frac{1}{5 \cdot 5^5} - \frac{1}{7 \cdot 5^7} + \dots \right) - 4 \left( \frac{1}{1 \cdot 239} - \frac{1}{3 \cdot 239^3} + \frac{1}{5 \cdot 239^5} - \dots \right)$$

### 4.12. Conversion of Number Format

The previous sections have described three methods of representing numbers in the machine storage. These are the binary fixed-point format, binary floating-point format, and packed decimal format. There is a group of arithmetic instructions in the machine for numbers of each format. To the above-listed formats we should add the zoned decimal format which is used to represent numeric information going from the man to the computer and from the computer to the man. In the zoned decimal format we use one byte for each decimal digit, the digit proper being in the low-order nibble, while the high-order nibble of the byte (a zone) usually contains hexadecimal F. An exception is the rightmost byte of the field occupied by the number. The number sign may be in this byte zone.

Here are several examples of representing numbers in the zoned format:

F	3	F	7	F	2	F	9	C	1	+37291
F	0	F	0	F	0	F	8			28 (unsigned number)
						D	5			-5

Data utilized in arithmetic operations of some types do not often correspond to the type of the arithmetical instructions used, for which reason they should be first converted to the required format. For example, data program-obtained in the zoned format may be converted to the packed decimal or any other format. Upon completion of the computations, before printing the results, reverse conversion generally takes place.

With the computers of the first and second generation, to perform any conversion one had to prepare a special subprogram similar to that given under 10, Chapter 4. The ES EVM instruction system includes four instructions performing the most often conversions.

The PACK instruction

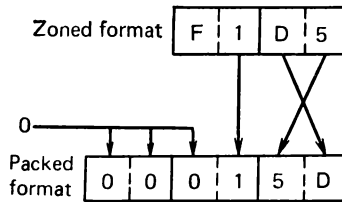
F2 PACK SS L<sub>1</sub>L<sub>2</sub>B<sub>1</sub>D<sub>1</sub>B<sub>2</sub>D<sub>2</sub>

The second operand is converted from the zoned format into the packed format and placed in the field defined by the first operand. The data are processed byte by byte, from right to left.

At the first step, the zone of the first byte of the second operand containing the sign code is placed in the low-order nibble of the first operand. Subsequently, the instruction handles only the right-hand



**Example.** Pack a two-byte field into a three-byte field:



If the first operand field is too large, its excessive positions are packed with zeros.

The UNPACK instruction

F3 UNPK SS L<sub>1</sub>L<sub>2</sub>B<sub>1</sub>D<sub>1</sub>B<sub>2</sub>D<sub>2</sub>

The UNPK instruction works just the opposite of the PACK instruction. The second operand is converted from a packed decimal format into a zoned format and is placed in the field defined by the first operand. The data are processed one byte at a time from right to left.

The low-order nibble of the packed format field containing the sign is placed into the low-order byte zone of the first operand field. All the other zones are assigned hexadecimal F's.

The decimal digits of the second operand field are inserted into the right-hand nibbles of the first operand field bytes. If the length of the first operand area is too large to store the result, the remaining bytes are packed with decimal zeros in a zoned format.

If the field of the first operand is too small to accommodate all data until the completion of processing the packed format field, the remaining unused high-order positions are truncated.

In the execution of the UNPK instruction, no check is made to see whether the field of the second operand contains a packed decimal number.

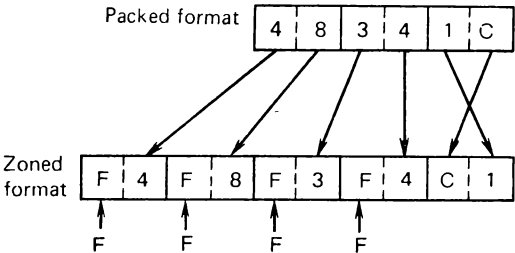
The maximum length of either of the operands is 16 bytes.

The condition code is not varied.

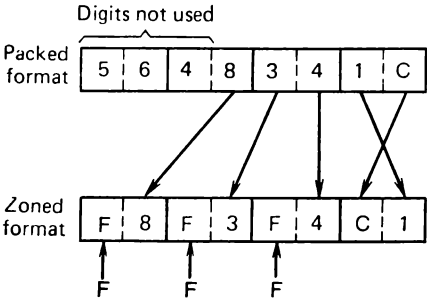
The length of the first operand field enough to store the result equals the length of the second operand field multiplied by two and decreased by 1. Thus, to unpack a three-byte field containing a decimal number in the packed format, use should be made of the first operand length equal to  $3 \times 2 - 1 = 5$  bytes.

The examples that follow illustrate various instances of the execution of the UNPK instruction.

**Example.** Unpack a three-byte field into a five-byte field:

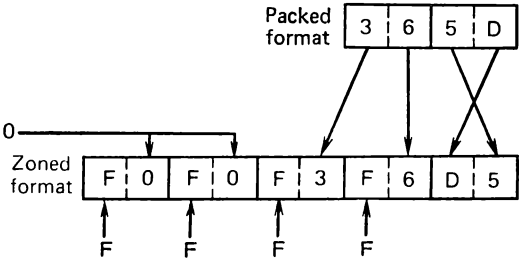


**Example.** Unpack a four-byte field into a four-byte field:



The high-order digits of the second operand that do not fit the first operand field are truncated.

**Example.** Unpack a two-byte field into a five-byte field:



The redundant bytes of the first operand field are packed with zoned format zeros.

The CVB instruction

CONVERT TO BINARY      4F CVB RX R<sub>1</sub>X<sub>2</sub>B<sub>2</sub>D<sub>2</sub>

The CVB instruction converts a packed decimal number defined by the second operand to a binary integer number, and places it in-

to general register  $R_1$ . The sign of the packed decimal number defines the sign of the binary number: if the sign nibble contains a hexadecimal B or D, then the decimal number is negative, and the binary number is represented in complement form. If the sign nibble contains A, C, E or F, the conversion result will be a positive binary number.

The packed decimal number should occupy a double word in storage, i.e. its length must be equal to 8 bytes, and the address of the leftmost byte of the number should be a multiple of eight.

The range of convertible numbers is limited only by the length of the general register. Therefore the source decimal number should lie within the limits from  $-2147483648$  to  $+2147483647$ . If a number is beyond these limits, the result will contain only 32 low-order bits of the binary number, and the execution of a program including a CVB instruction will be interrupted.

The condition code is not changed.

To place a packed decimal number in a double word, use may be made of a ZAP instruction.

The CVD instruction

CONVERT TO DECIMAL    4E   CVD   RX    $R_1X_2B_2D_2$

The signed integer binary number contained in general register  $R_1$  is converted into a packed decimal number and placed into the double word defined by the second operand. The sign code of the number is placed in the low-order nibble of the second operand field in compliance with the packed decimal format.

The condition code is not changed.

The PACK, UNPK, CVB and CVD may be used to manipulate numbers represented in various formats.

**Example.** The instructions given below add a zoned decimal number to an integer binary number. The sum is also in a zoned format.

*Before instruction execution*

Register 1	0000 80000
Storage 008000-05	F3F2 F1F4 F7C8
Storage 008006-07	FE74

The six-byte field at address 008000 contains a decimal number  $+321478$ . The halfword at address 008006 contains a fixed-point binary number  $-396$ . The sum is to be placed in a six-byte field at address 008008.

<b>Instructions:</b> PACK	F2 7 5 1 010 1 000
CVB	4F 2 0 1 010
AH	4A 2 0 1 006
CVD	4E 2 0 1 010
UNPK	F3 5 7 1 008 1 010

In this program the double word at address 008010-17 is utilized as a working area for temporary storage of packed numbers. General-purpose register 2 is used for storage of binary numbers and addition operation.

The PACK instruction converts number 321478 from a zoned to a packed format. As a result, the field at address 008010 takes the form:

Storage 008010-17      0000 0000 0321 478C

The CVB instruction converts the augend to a fixed-point binary format. The result of its execution is:

Register 2      0004 E766

After both numbers have been represented in a similar format, the addition can be carried out. The AH instruction adds the addend contained at address 008006 to register 2.

*The result is*

Register 2      0004 E63A

The CVD instruction converts the obtained sum to a packed decimal format. After its execution, the double word at address 008010 will contain

Storage 008010-17      0000 0000 0321 082C

The last instruction makes it possible to obtain a sum in a zoned format.

*The final result is*

Storage 008008-0D      F3F2 F1F0 F8C2

The above-listed instructions testify to the fact that

$$321478 - 396 = 321082$$

Note, that the program includes only one arithmetic instruction, while the other four instructions are used for converting the operands to a required format. This teaches us to thoroughly think over the techniques of representing data in machine storage. Thus, we may formulate a rule: *as far as practicable, use similar type data in calculations.*

### Exercises

1. Write down all possible methods of representing numbers in computer storage. There are special instructions described in this section for certain conversions from one method of representation to another. For the other conversions we have to write subprograms. An example of such a subprogram is given at 10, Chapter 4.

Write the standard subprograms for all possible types of integer conversion from one method of representation to another.

2. The position of the decimal point in a number represented in a zoned format may be specified explicitly by inserting the symbol 'point' (the hexadecimal code 4B) into the corresponding position of the field occupied by the number.

Write a standard subprogram for conversion of numbers with an explicit decimal point in a zoned format into a packed format with an implicit decimal point. The subprogram parameters are: the length of the field containing the source number, length of the result field, address of the source field, address of the result field, number of digits after the implicit decimal point in the result field. Think over a compact and convenient form of writing parameters for your subprogram.

3. Modify the program of Exercise 2 so that it will round off the result to a required number of digits after the assumed decimal point, if the number of fractional positions specified is too large.

4. Check your program for correct operation, if there is no decimal point in the source field (in this case, the source number is considered to be an integer).

5. Modify the program given in Exercise 4 as follows: if the source number fits the result field, then load return code 0 in general register 15. If the field specified for the result is too small and the high-order significant digits are truncated, then load return code 4 into register 15.

The return code may be analyzed by the calling program in order to see if the execution of the subprogram is successful.

#### 4.13. Edit Instructions

Internal representation of data in computer storage should be selected from the standpoint of the maximum efficiency of the operations performed. However, it is not often good to dump these data directly in that form in which they are represented in the computer, since the printed data will then be manipulated by a man, and we must ease his work as much as possible. With business-oriented computers a well thought out format of a printed document is often of more importance than, say, the operating time of the program.

The ES EVM instruction system includes two instructions simplifying reduction of numeric data to the form suitable for printing. Those are edit instructions referring to the means of processing decimal data.

The ED (IT) instruction

EDIT DE ED SS LB<sub>1</sub>D<sub>1</sub>B<sub>2</sub>D<sub>2</sub>

The source field defined by the second operand contains one or several numeric data which must be edited. The data contained in the

source field should be in packed decimal format. The number sign, if used, should be in the very rightmost nibble of the field, and all the other nibbles must contain correct decimal digits—hexadecimal digits from 0 to 9. If the source field contains several decimal numbers, each of them may contain the sign code in its right-hand nibble.

A source field containing one number

0	0	2	3	7	6	4	9	1	C
---	---	---	---	---	---	---	---	---	---

A source field containing several numbers

3	1	4	1	5	C	0	0	2	7	1	8	3	D	0	1	4	2	8	5	7	C
Number 1						Number 2						Number 3									

The length specified in the instruction applies only to the first operand. Prior to the execution of the instruction, an *editing pattern (mask)* must be placed in the first operand field. This pattern defines the necessary conversions according to the special rules. The result obtained after the editing operation is placed in the first operand field and fully replaces the pattern. Each byte of the pattern corresponds to one character being dumped.

The first byte of the pattern should contain an additional character called the *fill character*. A blank or an asterisk (\*) is usually used for the purpose, i.e. to suppress leading zeros—that is replace leading zeros with a blank, or an asterisk. The other bytes of the pattern may contain insert characters or special control characters which are any correct characters placed without changes in the edited field. An example is a decimal point.

The following three special control characters are used in the editing patterns:

Hexadecimal code	Name	Symbol
20	Digit selector	<i>d</i>
21	Significance starter	<i>s</i>
22	Field separation character	<i>f</i>

The *digit selector (digit select character)* is used to indicate a pattern position into which the next digit from the source field is to be placed. If the digit in question is a leading zero, then the digit selector is replaced with a fill character.

The *significance starter* (*significant start character*) performs the same function as the digit selector, but, in addition, it indicates that all the other digits (even nonsignificant zeros) must be considered significant. Therefore, with the aid of the significance starter the programmer may indicate that there is no need to suppress the leading zeros.

The *field separation character* is used in concurrent editing of several numbers and separates the pattern fields referring to each number. In the edited field, it is always replaced with the fill-character.

In addition to the source field and pattern, a special *significance indicator* showing that significant digits are present in a number participates in the execution of the editing instructions. The significance indicator is a two-state switch: 0 (OFF) and 1 (ON). The state of the significance indicator is set and checked in the course of editing to perform the required actions.

The editing involves analyzing the pattern one byte at a time, from left to right, and performing those actions on each byte which are dependent on the pattern character, successive digit of the source field, and state of the significance indicator.

Given below in detail is the algorithm of the execution of the ED instruction.

1. At the beginning of each editing operation the significance indicator is set to 0.

2. The fill character remains unchanged and is stored for future use.

3. Each time a digit selector  $d$  is encountered in the pattern, an analysis is made of the successive nibble of the source field. If this nibble does not contain a valid digit code (from 0 to 9), the execution of the instruction is interrupted because of invalid data.

4. If the digit selected is other than zero, a zone (hexadecimal F) is added to it, and the resultant character is placed in the result area to replace the pattern character. The significance indicator is set to 1.

5. If a successive digit of the source field is zero, the state of the significance indicator is checked. With the indicator in the off state (nonsignificant zero) the pattern character is replaced with the fill character, i.e. the zero is suppressed. The indicator in the 1 state indicates that either a significant digit has been found before, or the significance starter  $s$  has been encountered. In this instance, a zero is placed in the result area, as any other digit.

6. If a significance starter  $s$  is encountered in the pattern, the same actions as for the digit selector  $d$  are performed, but then the significance indicator is additionally set to 1.

7. If a field separation character  $f$  is encountered in the pattern, the significance indicator is set to 0 to allow leading zeros suppression for the next number from the source field.

8. When any character other than *d*, *s*, and *f*, appears in the pattern, the state of the significance indicator is checked. In case of the 1 state, the character in the pattern remains unchanged. In the 0 state it is replaced with the fill character.

9. During the execution of the instruction, the data from the source field are selected one byte at a time (two digits). The digit found in the left-hand nibble is processed first, after which checks are made at once to see whether the sign code is not in the right-hand nibble. If the right-hand nibble contains the sign code (a hexadecimal digit from A to F), then the processing of the number is completed, and there must be no digit selectors in the bytes remaining down to the end of the editing pattern (or to field separation character). If the number is negative, the significance indicator is set to 1, and these characters remain unchanged.

When a plus sign appears in the source field, the significance indicator is set to 0, and all characters down to the end of the pattern, or to a field separation character, are replaced with the fill character. Therefore, the editing instruction allows negative numbers to be marked.

10. Upon completion of editing, the following values of the condition code are set:

CC = 0, if the number being edited contains only zero digits.

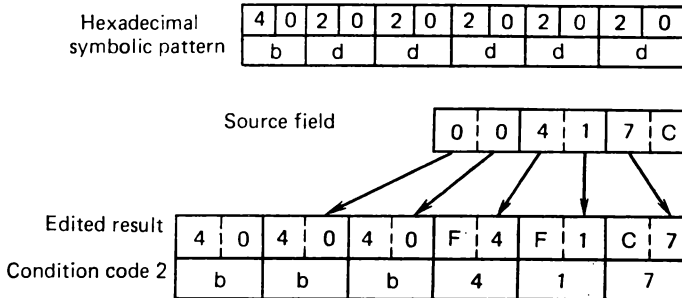
CC = 1, if the number being edited does not equal zero and the state of the significance indicator is set to 1 (the source number is negative and not equal to 0).

CC = 2, if the number being edited is not equal to zero and the state of the significance indicator is set to 0 (the source number is positive and is not equal to 0).

If the editing operation was performed concurrently on several numbers, then the value of the condition code refers solely to the last number.

Of all the ES EVM instructions, the editing instructions are most complicated, and perhaps it will be difficult for the reader to fancy at once all features and abilities of this operation. In the computers of the first and second generations, editing was performed by special fairly complicated subprograms comparatively slow in operation. The ES EVM designers have provided the programmer with a powerful tool of accomplishing this operation, especially effective in handling business-oriented information.

The remaining part of this section is dedicated to various examples of data editing. In the analysis of these sample examples the reader is recommended to return sometimes to the description of the instruction. This will help you understand all versions of the editing process. Note, that a blank is represented in the examples by the character *b*.

**Example**

This example illustrates the operation of suppressing leading zeros. Here we show in detail the structure of the editing pattern (mask) and the result field in a hexadecimal and symbolic form. Further, we shall confine ourselves to the description of the fields of the pattern and result in a symbolic form. To represent the fields in a hexadecimal form, substitute their hexadecimal codes for the symbols.

**Example**

Editing pattern (mask)	<i>bddbddd.dd</i>
Source field	1234 567C
Edited result	b12b345.67

In this example a decimal point is inserted into the number, and thousands are separated from hundreds by a space to make the reading easier. Dumped are 12 345.67

**Example**

Pattern	<i>bddbddd. dd</i>
Source field	0000 004C
Edited result	bbbbbbbb4

As a result of editing, the decimal point is replaced with a fill character, as there is no significant digit to the left of the point. To cancel leading zeros suppression, use should be made of a significance starter. The corresponding pattern (mask) will be as follows:

*bddbdds.dd*

as a result, the edited field will take the form

*bbbbbbb.04*

Note once more, that the significance starter stops leading zeros suppression in the subsequent positions rather than in the current position. If you wish to print the result in the form 0.04, shift the *s* one position to the left: *bdbdsd.dd*.

**Example**

Pattern	<i>*dddsd.dd</i>
(a) Source field	0025 475C
Edited result	<b>***254.75</b>
(b) Source field	0000 046C
Edited result	<b>****0.46</b>

In these examples asterisks are inserted in all positions of result, close to the first significant digit, or to the place where the significance indicator has been set to the ON state by the significance starter. Editing of this type (with asterisks) is generally used in printing financial documents with a view to prevent forgery. For example, an amount printed as

547

can easily be changed to

99547

whereas

**\*\*547**

could be altered less easily.

**Example**

Pattern	<i>bds.dddb —</i>
(a) Source field	0237 4C
Edited result	<i>bb2.374bb</i>
(b) Source field	0005 8D
Edited result	<i>bbb.058b —</i>

After editing with the aid of the pattern specified in the example, negative numbers will be minus signed.

**Example**

Pattern	<i>bdbdddb pyб bddb коп</i> (pyб means roubles, коп stands for copecks)
(a) Source field	6250 94
Edited result	<i>b625b0b pyб b94b коп</i>
Dumped is	6 250 pyб 94 коп
Condition code	1
(b) Source field	0000 32
Edited result	<i>bbbbbbbbbb32b</i> коп
Condition code	1
(c) Source field	0000 00
Edited result	<i>bbbbbbbbbbbbbb</i>
Condition code	0

As a result of the editing, the numeric values are supplemented with text explanations. In this example unsigned packed fields are

edited and the text after the last digit of the number will not be replaced with blanks, as the case was with the previous example.

The condition code after editing an unsigned decimal field is set as follows:

CC = 0, if all the digits of the source field are zeros (the number equals zero).

CC = 1, if other than zero digits are in the source field (in this case, the significance indicator is set to 1)

### Example

Pattern	<i>bdddfddds.dfs.d</i>
Source field	075C 0443 8200 6D
Edited result	<i>bb75bb4438.2bbb.6—</i>
Condition code	1 (it is set according to the last digit)

In this example three decimal numbers are edited concurrently. The results do not differ from those that can be obtained by editing these numbers separately. It should be only noted that in editing several fields, the fill character is similar for each of these fields. It is the first character of the entire pattern.

The EDMK instruction  
EDIT AND MARK DF EDMK    SS LB<sub>1</sub>D<sub>1</sub>B<sub>2</sub>D<sub>2</sub>

This instruction performs all functions of the ED instruction. Besides, the address of the pattern byte containing the first significant digit of the source number is placed in general register 1. The address is loaded in the three low-order bytes of register 1, while the leftmost byte remains unchanged.

If the significance indicator sets to 1 because a significance starter appears in the pattern, no address is loaded in register 1 and the previous content remains in this register.

The EDMK instruction is used to insert characters allowed to float into an edited field.

### Example

The program segment given below edits a decimal number. If the number is negative, a minus sign is placed close to the leftmost significant digit in the result field.

Register 3	0000 4000
Register 10	0000 8000
Storage 004400-0C	Field for edited result
Storage 00440D-11	Source decimal number
Storage 00412-1A	4020 4020 2020 4020 2021 4B2020
	(pattern <i>bdbdddbdds.dd</i> )

## Program instructions

008000-05	MVC	D2	0C	3	400	3	412
006-09	LA	41	10	3	40A		
00A-00F							
010-013	EDMK	DF	0C	3	400	3	40D
014-015	BC	47	A0	A	01A		
016-019	BCTR	06	10				
01A-...	MVI	92	60	1	000		
	...				...		

The MVC instruction moves the pattern to the result field (to perform editing directly in the pattern field is inconvenient, as the pattern is spoiled in that). The next instruction LA loads the address of the result field byte containing the decimal point in register 1. If no significant digit is found before a significance starter appears in the source number, then, in editing, the address is not placed in register 1. In this case, let us insert the sign directly before the decimal point.

The BC instruction checks the condition code after the editing. If  $CC = 0$  or  $CC = 2$ , i.e. the number being edited is not negative, the instructions of entering the sign are omitted.

The BCTR instruction subtracts 1 from the content of register 1. In this case no branch occurs, as in this instruction  $R_2 = 0$ . As a result, register 1 now contains the address of the byte immediately preceding the first significant digit (or the decimal point). The next MVI instruction enters at this address a byte having the hexadecimal value of 60 — a minus sign code.

Below, the right-hand column contains the results of the work performed by this program for various source values given in the left-hand column.

1234 5678 9D	—1b234b567.89
0014 2857 1C	b b b b 14 b 285.71
0000 0000 0C	b b b b b b b b b .00
0000 0000 7D	b b b b b b b b —.07

The EDIT instructions give the programmer wide possibilities in representing numbers in the required format.

**Example**

1. Construct a flow-chart for the algorithm of executing the ED instruction, using the material dealt with in this section.

2. Construct a flow-chart for the algorithm of executing the EDMK instruction. Compare it with the flow-chart for the algorithm of executing the ED instruction. Modify, if necessary, both flow-charts to minimize their discrepancies as far as possible.

3. Suppose that the ES EVM instruction system does not include ED and EDMK instructions. Prepare standard subprograms per-

ming all functions of these instructions. Use the flow-charts constructed by you. Think over a call of your subprograms which is most convenient for the user.

4. Compare both programs given in Exercise 3. They are much alike. Modify the program so that it can perform the functions of both instructions, depending upon the call.

**IMPORTANT!** The program may be given as a parameter a switch defining the type of editing. This, however, is not the best solution. It is better to prepare one subprogram with two entry points. The type of editing is defined according to the program entry point. Special precautions should be taken to properly set the base register regardless of what entry has been used for the call.

5. Register 0 contains a fixed-point binary number, and register 1—the number of characters in this number after the default binary point. Prepare a standard program which prepares a number for printing. The program must define the number of decimal digits after the point which must be saved to provide the same precision as that of the source number. Then the program must convert the number into decimal notation and edit it by shifting to the right-hand end of the 16-byte field. The left not used bytes of the field should be filled with blanks, the decimal point must be inserted in the result. As the next step the program must suppress the leading zeros and place a sign before the first significant digit.

The address of the result field should be placed in register 1.

#### 4.14. Translation Instructions

This section deals with two instructions facilitating the symbolic information processing.

The TR instruction

TRANSLATE DC TR SS LB<sub>1</sub>D<sub>1</sub>B<sub>2</sub>D<sub>2</sub>

The length specified in the instruction applies only to the field of the first operand, containing the source symbolic information. The address of the second operand defines in the main storage the source byte of the 256-byte *translation table*.

The TR instruction operates as follows.

1. It fetches the data byte specified by the first operand of the instruction;

2. The content of this byte considered as an integer eight-bit binary number is added to the effective address of the second operand:  $E_2 = (B_2) + D_2$ ;

3. Using the address found a byte is selected from the table and substituted for the data byte used to find it.

The above-described actions are carried out in sequence for each byte from the field of the first operand.

In the execution of the TR instruction the condition code remains unchanged.

The TR instruction is used for examination of symbolic data and replacement of characters contained in the source field with others. The replacement rules are tabulated to form a translation table: the initial byte of the table is substituted for all characters of the source field which have hexadecimal code 00, the next (second) byte—for characters with code 01 and so on. The character of hexadecimal combination FF, wherever it may be encountered in the source field, must be replaced with the last 256th byte of the table.

The TR instruction finds its main applications in conversion an eight-bit code for representation of symbolic information into another. Such actions may be necessary when together with the computer use is made of a device operating in another than the EBCDIC code accepted in the ES EVM family of computers. If, for instance, the results of the computer work will be transmitted by a teletype to another city, the information should be translated to the MTK-2 telegraph code.

Sometimes, the TRANSLATE instruction is used for editing symbolic data before printing. Often in addition to the information the fields being processed contain various service symbols. The TR instruction may be used for, say, replacing these symbols before printing with blanks.

The TRT instruction

TRANSLATE AND TEST DD TRT SS LB<sub>1</sub>D<sub>1</sub>B<sub>2</sub>D<sub>2</sub>

Like in the TR instruction, the address of the second operand defines a 256-byte table. The length specified in the instruction, applies only to the field of the first operand. During the execution of the instruction, the field of the first operand is examined in sequence, starting with the left-hand byte, and the following actions are performed on each byte.

1. Fetch is made of the data specified by the first operand of the instruction.

2. The content of this byte considered as an integer eight-bit binary number is added to the effective address of the second operand:  $E_2 = (B_2) + D_2$ .

3. According to the address found, the byte is fetched from the table, but unlike the TR instruction, the byte of the first operand is not changed.

4. If the byte fetched from the table equals zero, the next byte is fetched from the field of the first operand, otherwise the address of the byte from the first operand field is loaded in the right-hand 24 bits of general register 1, while the byte fetched from the table is placed in the first eight bits of general register 2. The left-hand eight

bits of register 1 and left-hand 24 bits of register 2 remain unchanged. This terminates the execution of the instruction.

The execution of the TRT instruction results in the following condition code settings:

CC = 0—the whole of the first operand field has been scanned and all bytes fetched from the table are zeros.

CC = 1—a non zero byte is encountered in the table before completion of scanning the field of the first operand.

CC = 2—last byte fetched from the table turns out to be other than zero.

The actions performed by the TR and TRT are alike, but their fields of application are completely different. The TR instruction is suitable for replacing the source field characters, while the TRT instruction is good for search of specified characters in the source field. For instance, the TRT instruction is easily used to check a certain sequence of characters, fetching those bytes in it which are necessary for processing. These may be invalid characters, blanks, commas, etc. The table for the TRT instruction is built up as follows: the characters of the input sequence (string) in question are collated with non-zero bytes of the table. All the other bytes must be equal to zero.

#### 4.15. The Execute Instruction

The EXECUTE instruction refers to the branch instructions, since it disturbs the natural flow of the instructions being executed. However, it is a branch of unique nature:

EXECUTE    44   EX   RX    $R_1X_2B_2D_2$

The address of the second operand defines the instruction which is to be executed, and, like the address of any other instruction, should point to the boundary of a halfword. The instruction being executed is sometimes called a subject instruction. Only bits 24-31 (the rightmost byte) are used in general register  $R_1$ .

The EXECUTE instruction causes the execution of a single instruction that is out of the normal instruction sequence, except another EXECUTE instruction. When an EXECUTE instruction is encountered, the address of the second operand is calculated, and the instruction at this address is fetched and executed in a normal way.

If the subject instruction is a branch instruction, then the branch is made as usually. In all other instances the next instruction after the EXECUTE instruction will be executed.

The above actions are accomplished, if the first operand of the EX instruction is general register 0. If  $R_1 \neq 0$ , then before execution of the subject instruction, the rightmost byte of register  $R_1$  is logically one byte at a time added to the second byte of the subject instruction.

Thus, a portion of the instruction is modified (certainly, the instruction in the storage remains unchanged, modified being only its effective kind).

In the instructions of the RR, RX and RS types, the modified byte contains registers, in the instructions of the SI format this byte specifies immediate operand  $I_2$ , and in the instructions of the SS format it indicates the length of the operands.

The EXECUTE instruction is generally used when information contained in the modified byte of the subject instruction is unknown during writing of the program and becomes available only during its execution.

An example may be a subprogram processing text messages of variable length. If in the course of operation it becomes necessary, say, to move a message to the working field, then it is convenient to execute the MVC instruction with the help of the EX instruction. Specified in the MVC instruction itself is a zero length indicator, the required number of bytes are moved by specifying a message length in the low-order bits of register  $R_1$  of the EX instruction.

### Exercises

1. The machine storage contains a string of characters including decimal numbers. The numbers are written in the following format:

The first character in the field occupied by the number may be a sign character: + or — character; if no sign is present, the number is positive.

The sign is followed by several decimal digits in a zoned format, and, perhaps, by the decimal point which separates the fractional portion from the integer portion; if no decimal point is used, the number is a whole number.

A colon (:) is used to separate the numbers from each other.

The end of information in the source field is marked by an asterisk (\*) found in a successive byte.

Design a standard subprogram which scans (examines) this string of characters, converts the numbers into packed decimal form at rounded to three digits after a default decimal point, and places them in sequential six-byte fields.

The addresses of the storage areas containing the source information which are intended for placing the result are transferred to the subprogram as parameters. The quantity of the formed numbers must be placed by the subprogram in general register 0.

2. Modify the subprogram described in Exercise 1 so that it will round off numbers to three digits after the decimal point when the source number contains more digits in the fractional portion.

3. Have you utilized TRT and EX instruction? If not so, re-write the subprogram given in Exercise 1 so that these instructions are used.

4. What will you do, if there are errors in the source information for the subprogram used in Exercise 1? Mention all possible types of errors and modify the program flow-chart respectively in order to form only valid information in the source field. Count the total number of errors and place the result in the left-hand half of general register 0. Make the corresponding corrections in the program.

## CHAPTER 5

## CONTROL

### 5.1. Control Programs

The previous chapter dealt with various information handling instructions. This chapter covers another, far smaller, but no less important group of instructions which are used to control the computation process.

What is the purpose of these instructions? One can visualize the following scheme of computation organization: a programmer loads a program in the machine, keys the address of the program first instruction on a console made for the purpose, and depresses the START push-button. The computer starts its operation. Upon completion of the calculations, the computer stops, the programmer takes the results of computation, and gives up the programmer's place to the next user. Halts occur when the execution of the program cannot be continued without manual intervention. Examples are end of paper on the printer, or an instruction with an invalid operation code encountered in the program (such halts are called *abends*). The man at the console of the computer must determine why the *abend* has occurred and, after that, take measures to continue the computation work. If the program fails to be restarted, a decision is made to terminate the work on a given program.

The above technique of computation organization is used on all computers of the first generation and many computers of the second generation. The operating speed of these machines is low, and the execution of programs takes, on the average, much time—tens and hundreds of minutes, and several minutes may be well allowed to change over from a program to another, or to find out the cause of the computer trouble.

With the growth of computer speed, the situation radically changes. The modern powerful computers are capable of performing hundreds of thousands and millions of operations per second. Under such conditions the average time of one program run is cut down to several tens of seconds. Note for illustration that a minute of run on a

computer having an operating speed of 1 000 000 operations per second is equivalent to about three and a half hours of computations on a computer whose speed is 5 000 operations per second.

However, the 'operating speed' of a man remains unchanged, and as before manual operations need several minutes each. Therefore, with the previous organization of computer work, the costly computer would idle most of its operating time. With the growth of computer operating speed, the man becomes a 'bottle-neck', and the only solution of this problem is to make the computer itself responsible for the whole of work organization, leaving the man only servicing functions such as to mount a successive deck of cards or spool of magnetic tape on an appropriate device, unload the output devices and forward the computation results to the user, etc.

A set of programs which organize continuous operation of a computer without manual intervention is a necessity of today's computers. Such a set of programs is known as an *operating system*. One of the operating systems designed for the ES EVM is a DOS/ES described in a condensed form in Chapter 10. We are here interested in the operating system only from the standpoint of those hardware facilities which are necessary for the system functioning.

The central part of the operating system is a *master control program* under the control of which the computation work is accomplished. The master control program starts the users' programs and the programs included in the operating system (*system programs*), changes over from a program to another, handles fault and other abnormal situations, and performs many other functions. As to its structure the master control program consists of three main parts.

**1. The initial system loader.** When the machine starts its operation, a special hardware-implemented procedure known as the *initial program loading* (IPL) loads a program which performs certain functions necessary for system operation, and then enters the *supervisor*.

**2. The supervisor.** This is a permanently stored or *resident* portion of the master control program. As it follows from its name, the supervisor functions consist in the control of current operation of the computer. The supervisor provides the standard handling of all particular situations in the programs, controls all *input/output operations* (for the information on input/output operations, see the next chapter), performs certain housekeeping functions on requests of the programs, etc.

In the memory the supervisor is arranged in the area with low-order addresses where it occupies several thousands of bytes. The actual size of the supervisor is dependent upon the actual set of input/output devices and the operating system release used.

**3. The job control.** When a program terminates its work (normally, or abnormally), control is passed to the supervisor which loads the

*job control program* into the memory. This program reads the description of a new job from the standard reader unit (this is generally a card reader), tests it, and calls in a required program. All actions are carried out automatically, without intervention of the man whose only duty is to load decks of cards containing new jobs in due time.

The instructions considered in the previous chapter are not sufficient to control the computation process. This chapter deals with additional facilities necessary for successful operation of the control program.

## 5.2. Supervisor Work Supports

As it has been said, the high-speed computer should not stop to wait for a decision of the man. Halts are allowed only in two instances: if there is no work, or when a trouble occurs. In all other instances of particular situations in the programs and upon completion of each program, control is passed to the supervisor which takes all the steps necessary to continue the work.

The supervisor, however, cannot receive control like an ordinary program block, as a result of a branch instruction execution. The supervisor intervention may be required at any unpredictable time, say, in case of an error encountered in a user's program, or when an input/output device has completed its operation, and appropriate measures should be taken. Therefore, to support the operation of the supervisor use should be made of a special control transfer tool known as *interruption*.

Briefly, the interruption concept may be explained as follows. The computer storage simultaneously contains two programs: the working storage used to perform the computation, and the supervisor—a program used to handle special situations. Generally control rests with the working storage and its instructions are carried out one after another. However, at any time, as soon as the machine circuits detect an unusual situation, a special *interruption signal* is produced, which makes the CPU stop its current operation regardless of what it is doing, and control is passed to the supervisor. The address of the first instruction of the supervisor program is taken from a special storage location which is used solely for this purpose. Upon receipt of control, the supervisor determines the cause of interruption and performs the actions required, and then either stops the execution of the application program, or returns control to it to continue the computation work.

For the first time, the interruption concept was used in development of certain computers as far back as 50s. However, it was only in the third generation computers that a well-developed interruption aid became a necessary component of the computer structure. As a matter of fact, interruptions are the only effective and convenient

tool of linking the control program to the computation process.

Like in any instance of passing control from program to program, the problem of saving the state of the interrupted program must be solved. However, in this case, in addition to the registers whose state is saved by the supervisor in a usual way, extra information, for instance, the condition code, or the address of the instruction being executed at the time of interruption should be stored. If after the operation of the supervisor, you know, the execution of the application program is continued, the exact state of the CPU the instant the interrupt occurs, should be recovered.

In the ES EVM all information about the state of the processor is contained in the *program status word* and its save operations are accomplished automatically during interruption. In order to continue the processing according to the program interrupted, the supervisor recovers the stored PSW referring to the interrupt instant.

The next facility necessary for successful operation of the supervisor is *system protection*. Programs executed by a computer contain errors at least in the debugging stage. These errors may result, say, in an attempt of the program to write some information in the storage area occupied by the supervisor. If no protection means are used, the supervisor would be destroyed, and the system would have to be given initial loading once more. When no protection features exist, errors in programs can involve more aggravated effects. An example may be an erased program library stored on a magnetic disk or magnetic tape.

The following principle underlies the system protection: the program can perform any operations within the limits of the storage area assigned to it, because errors in this case will affect only this program, while the operations needing any means in addition to the assigned storage area and registers, are either disabled at all, or, if necessary, may be carried out only on a permit of the supervisor which determines whether the operation requested is legal and safe. Two conclusions may be drawn from this principle.

1. The data processing instructions should not perform operations on storage fields beyond the segment assigned to the program, i.e. the machine design must provide hardware-implemented *storage protection*.

2. Execution of instructions requiring any facilities, except for the storage and registers, for instance, instructions setting storage protection, I/O instructions, etc. must be disabled in the application programs. Such instructions are called *privileged instructions* and can be executed only by the supervisor.

In the end, it may be said that the supervisor operation requires:

- A developed facility of interruption, including automatic storage of the CPU state;
- Storage protection facilities;

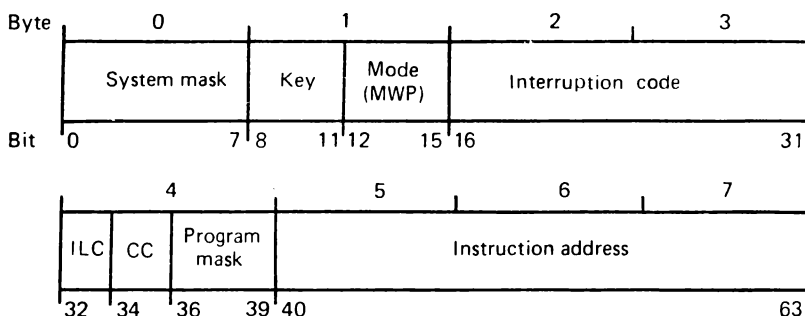
— Use of privileged instructions.

In addition, it should be noted that it is good to have a simple facility by means of which the program can reference the supervisor. A reference to the supervisor may be required, for instance, to tell it about completion of the program work.

The sections of this chapter that follow describe how these requirements have been satisfied in the ES EVM.

### 5.3. Program Status Word (PSW)

The program status word (PCW) is 64 bits in length (a double word) and has the following format:



Bits 0-7 contain the *system mask*. It refers to the information input/output and will be described in the next chapter.

Bits 8-11 contain the *storage protection key*. The storage segment assigned to the program is also provided with the same protection key, and in execution of any instruction it is determined whether the key in the PSW corresponds to the keys of the storage fields (2K blocks) utilized in the instruction. The storage protection is considered in detail in Sec. 5.4.

Bits 12-15 are switches setting various *CPU modes*.

Bit 12 defines the *code for information exchange* which is utilized in the ES EVM. If this bit contains zero, the computer operates in the EBCDIC code. With a 1 in bit 12 of the PSW, the computer operates in the ASCII-8 code. The latter code is used, but very seldom, and bit 12 of the PSW generally contains zero.

Bit 13 equal to 1 means that interruptions occur because of *hardware checks* (failures), if hardware malfunctioning is encountered. With a zero in this bit, hardware faults are ignored. Generally, bit 13 of the PSW is one, provided no special repair or adjustment operations are being carried out.

Bit 14 being set to one defines the CPU wait state. If this bit is set to zero, the processor is in a *processing state* in which the instructi-

ons are processed in the normal manner, one by one. In the *wait state*, the instructions are not executed, but interruptions occur in either of these states in a similar way. The supervisor sets bit 14 of the PSW to one, if the program cannot continue its work till the next interruption. This is the case, when the program, say, is awaiting completion of an input/output operation.

Bit 15 of the PSW set to one means that the CPU is in the *problem state*. The CPU is in the *supervisor state* when bit 15 of the PSW is zero. In this state, all computer instructions are valid. With the CPU in the problem state, the privileged instructions are disabled.

Bits 16-31 contain an *interruption code*. When any interruption occurs, a certain code is placed into these bits before the old PSW is stored. Analyzing the interruption code, the supervisor readily determines the cause of interruption and takes the corresponding steps.

Bits 32-33 contain an *instruction length code* (ILC) that indicates the number of halfwords in the last instruction being executed: 01—for the instructions of the RR format, 10—for the instructions of the RX, RS and SI formats, and 11—for the SS type instructions. Code 00 indicates that the instruction length has not been defined by the computer. This is the case, when, say, an instruction is encountered having an invalid operation code.

Bits 34-35 contain the *condition code* (CC).

Bits 36-39 contain the *program mask*. It defines, whether certain special situations (an example is a significance exception) will cause an interrupt or will be ignored. The program mask is considered in detail in Section 5, Chapter 5.

Bits 40-63 contain the *address of the next instruction to be executed*.

Each program being processed requires the PSW which contains all information about the CPU state which refers to the program, except the contents of the registers.

When an interrupt occurs, an *interruption cause code* is placed in the current PSW, and then the PSW is stored in a double word of the main storage which is provided for the purpose. The new PSW is taken from another double word, and the processor uses it to start the supervisor block handling interrupts of this type. The ES EVM provides five types of interrupts, with each of which two double words are associated: one for storing the old PSW and one for storage of the new PSW. The addresses (locations) of these double words are fixed in main storage as follows:

<i>Type of interruption</i>	<i>Old PSW location</i>	<i>New PSW location</i>
External	18	58
Supervisor call	20	60
Program	28	68
Machine check	30	70
Input/output	38	78

All addresses (locations) are in the hexadecimal notation.

*External interrupts* can come from one of three sources:

1. The *timer interrupt* occurs when the time keeping device (a clock) built into the computer has counted off a specified interval of time. The timer is employed by the supervisor to account for time taken by execution of various jobs. Besides, on requests of the programs, the supervisor may give current time, say, for tying computation results to it. The time counting is started with the IPL execution.

2. The *interrupt from the console* is caused by depressing a special button INTERRUPT on the computer control console. This facility may be used by the machine operator to attract the supervisor attention to receive operator's directives or promptings.

3. The *interrupt from peripheral devices* occurs when a device, external with respect to the computer, sends a predetermined signal. This type of interruption is used to organize two-way communication with peripherals.

The *supervisor call interrupt* is considered in Sect. 5.6.

The *program interrupts* arise from encountering errors in the program, such as an invalid operation code, nonexistent storage address, and also in particular situations—zero divides, invalid digit code in decimal arithmetic instructions, etc. For the program interrupts in detail, see Sect. 5.5.

The *machine check interrupts* result from a hardware malfunction. If that is the case, the computer usually needs repair or adjustment.

The input/output interrupts are dealt with in the next chapter.

If, upon completion of an interrupt handling, necessity arises to continue the program interrupted, the supervisor must restore the old PSW stored in the corresponding double word. To this end, the instruction system includes a special instruction LOAD PSW.

LOAD PSW    82   LPSW   SI   I<sub>2</sub>B<sub>1</sub>D<sub>1</sub>

The second operand I<sub>2</sub> is not used. The address defined by the first operand should point to the boundary of the double word containing the PSW to be loaded. The instruction at the address contained in bits 40 through 63 of the new PSW will be executed next.

The LPSW instruction is privileged, i.e. it may be executed by the processor only in a supervisor state.

#### 5.4. Storage Protection

The storage protection in the ES EVM includes a *protection key* contained in each PSW, and *storage keys* provided in each 2048-byte block of main storage. The storage key consists of eight bits of which bits 0-3 contain the key itself, bit 4 defines the *protection mode*, and bits 5 through 7 are always equal to zero.

If bit 4 of the storage key is zero, then the corresponding storage block is *protected against store*. Any attempt of the program to store information in this block is accomplished according to the following rules. If the protection key in the PSW is equal to the content of bits 0 through 3 of the storage key, or the protection key in the PSW is zero, a store operation is permitted. Otherwise, no store is performed and storage protection *program interrupt* occurs. However, with bit 4 of the storage key set to zero any program may read any storage location of this block, regardless of whether the keys match, or not. In this case, it is said that the storage block is protected against store, but *not protected against fetch*.

If bit 4 of the storage key equals one, then with the protection key in the PSW not matching the storage key, neither a store operation into a given block of storage, nor fetch of any information from it is permitted. Therefore, the storage blocks can be protected solely against a store operation, or against store and fetch operations together.

Since the supervisor should have access to any storage location, all PSWs of the supervisor have a zero protection key. All storage areas utilized by the supervisor also have a zero storage key and are not protected against a fetch operation. All the other programs are assigned nonzero protection and storage keys.

Each storage block is given a key, regardless of the other blocks. Therefore, a program may occupy several blocks which may not be adjacent. To manipulate the protection keys the ES EVM system of instructions includes the following two instructions:

SET STORAGE KEY    08   SSK   RR    $R_1R_2$

Bits 8 through 31 of general register  $R_2$  must contain the address of the storage block the key is assigned to. The block address is a multiple of 2048, and the content of bits 21 through 31 is of no importance. However, the algorithm of instruction execution requires that the low-order four bits (bits 28 through 31) of register  $R_2$  contain zeros. The value of bits 0 through 7 (the left-hand byte) of register  $R_2$  is ignored. Bits 24 through 31 of register  $R_1$  (the right-hand byte) must contain the storage key. The value of bits 0 through 23 and 29 through 31 of register  $R_1$  is ignored. Bits 24 through 28 of register  $R_1$  are placed in bits 0 through 4 of the storage key, and bits 5 through 7 of the storage key are set to zero.

INSERT STORAGE KEY    09   ISK   RR    $R_1R_2$

Like in the SSK instruction, the address of the 2048-byte storage block is specified in  $R_2$ . The corresponding storage key is placed in the rightmost byte (bits 24 through 31) of general  $R_1$ . The other bits of register  $R_1$  remain unchanged.

The SSK and ISK instructions can be executed by the processor only in the supervisor state.

### 5.5. Program Interrupts

Nearly all ES EVM instructions impose certain restrictions on the addresses of the operands and data the operation is performed on. For instance, the address of the operand pointing to a word in main storage must be a multiple of four. The size of a quotient resulting from fixed-point division should not exceed 31 bits, etc. If this requirement is not satisfied, the instruction will not be executed at all, or will produce an invalid result.

During execution of a program the supervisor constantly manages the observation of all requirements necessary for proper execution of the instructions. Program interruption results from any of the abnormal situations described below. If that is the case, the interruption code is placed in bits 16 through 31 of the old PSW which is stored at address  $28_{16}$ . Control is passed to the supervisor block handling program interrupts whose address is defined by the new PSW loaded from location  $68_{16}$ .

Listed below are 15 events causing program interrupts with indication of the interruption codes and their causes.

*0001 Invalid operation code.* An instruction having a nonexistent code of operation is encountered in the flow of instructions.

*0002 Privileged operation.* An attempt to execute a privileged instruction in the *problem* state. The operation is not executed.

*0003 Invalid EXECUTE instruction.* An EXECUTE instruction led to another EXECUTE instruction (see Sect. 4.15).

*0004 Disturbances to storage protection.* The storage key of a 2048-byte block whose location is called does not match the protection key in the PSW. The data reference to which has caused the protection trouble are neither stored, nor fetched from storage. If the protection trouble is caused by an attempt to execute an instruction beyond the area allocated for the program, the instruction is not processed and in this case the ILC (the instruction length code) is set to zero.

*0005 Invalid addressing.* This interrupt arises in an attempt to call a storage location not present on a given machine. The operation is stopped. When an attempt is made to execute an instruction from a nonexistent location the operation is not executed.

*0006 Specification—incorrect word boundary or register.* This type of interruption occurs in the following cases:

— The address of the instruction is not a multiple of two; address of a halfword is not a multiple of two; address of the word is not a multiple of four; address of a double word is not a multiple of eight;

— In the instructions using two registers (an even and an odd), a register having an odd number is specified in field  $R_1$ ;

— In floating-point instructions the address of a floating-point register is not equal to 0, 2, 4, or 6;

— In instructions of decimal multiplication (division) the length of the multiplier (divisor) exceeds eight bytes, or the length of the first operand field is no greater than the length of the second operand field;

— The content of the four low-order bits of register  $R_2$  in instructions SSK and ISK is not equal to zero.

In all these cases the operation is not performed.

0007 Invalid data. This type of interruption occurs in the following instances:

— In operations of decimal arithmetic, editing, or conversion to binary form, an illegal code of a digit or sign is encountered in a packed decimal number;

— In decimal arithmetic operations fields overlap improperly;

— The multiplicand in a decimal multiplication instruction does not contain at least  $L_2 + 1$  zero high-order bytes.

Execution of an instruction is stopped as soon as a data error is detected.

0008 Fixed-point overflow. A fixed-point overflow occurs, if a carry is made into a sign bit in fixed-point arithmetic or arithmetic shift to the left instructions, or when a high-order significant bit of the result is missing. The operation is terminated, the information that has gone beyond the register limits is lost, and an interruption occurs. Bit 36 of the PSW (of the program mask) masks this interrupt. If this bit is set to one, the interrupt is permitted. No interrupt occurs when bit 36 of the PSW is set to zero, and the overflow is ignored.

0009 Incorrect fixed-point divide. This type of interruption occurs, when result of fixed-point divide, or result of a CVB instruction occupies more than 31 bits. No divide is performed. Execution of the CVB instruction is terminated, but the information beyond the register boundaries is lost.

000A Decimal overflow. A decimal overflow occurs if in execution of any decimal arithmetic instruction the result does not fit the field assigned to it. The operation is always completed, the excessive high order significant digits being lost. Bit 37 in the PSW is intended for masking this interrupt: if this bit is set to one, the interrupt is permitted, otherwise the interrupt is ignored.

000B Incorrect decimal divide. This interrupt occurs if a decimal divide quotient does not fit the field assigned to it. The operation is not executed.

000C Exponent overflow. This type of overflow is encountered in floating-point instructions if the result characteristic exceeds 127. The operation is completed after subtracting 128 from the characteristic.

*000D Exponent underflow.* The result of a floating-point operation has a negative characteristic (the exponent is less than  $-64$ ). If bit 38 in the PSW masks the interrupt, the operation is terminated by issue of a zero result. If bit 38 in the PSW is set to one, 128 is added to the characteristic, the operation is completed, and an interrupt occurs.

*000E Significance exception.* The result of a floating-point operation has a zero mantissa and a nonzero characteristic. This interrupt does not occur in execution of the HALVE (HDR and HER) instructions. Bit 39 in the PSW masks this interrupt: if this bit is set to zero, the operation is terminated by issue of a zero result (with zero characteristic) and no interrupt occurs. Otherwise, the result characteristic remains unchanged and an interrupt occurs.

*000F Incorrect floating-point divide.* This type of interruption arises when the divisor in a floating-point divide operation has a zero mantissa. No operation is executed.

Four program mask bits in the PSW mask four of the fifteen listed program interrupts. Situations resulting in those interrupts may be encountered in programs of correct work. In that, their occurrence often involves actions other than the standard response of the supervisor to program interrupts. If that is the case, the corresponding interrupt must be masked to prevent supervisor intervention, and an occurrence of an abnormal situation can be traced by monitoring the condition code.

Situations leading to the other eleven program interruptions always indicate an error in the program, so that their masking is of no reason.

For setting a program mask the ES EVM system of instructions includes a special instruction

SET PROGRAM MASK    04   SPM   RR    $R_1R_2$

Bits 2 through 7 of general register  $R_1$  are substituted for bits 34 through 39 of the PSW. The other bits of register  $R_1$  and the whole of content of register  $R_2$  are ignored. Bits 2-3 of register  $R_1$  are substituted for the condition code and bits 4 through 7 for the program mask. The remainder of the PSW is not changed. At this point let us also consider in more detail the BRANCH AND LINK instructions

BRANCH AND LINK    05   BALR   RR    $R_1R_2$

BRANCH AND LINK    45   BAL    RX    $R_1X_2B_2D_2$

Low-order 32 bits of the PSW which contain the instruction length code, ILC = 01 for BALR, or ILC = 10 for BAL, condition code, program mask and address of the instruction following BALR or BAL are loaded in general register  $R_1$ .

A branch is made on the effective address  $E_2 = (X_2) + (B_2) + D_2$  for the BAL instruction either at the address contained in the three right-hand bytes of register  $R_2$ , run of the BALR instruction. If use is made of a BALR instruction in which  $R_2 = 0$ , no branch occurs and the next in sequence instruction is executed.

Therefore, the BRANCH AND LINK instructions do not only store the address of the point of return to an externally stored program, but also save the value of the program mask utilized in the externally stored program.

If a subprogram needs other modes of handling masked interrupts, the values of the program mask bits must be modified. Generally, the content of the return register is moved to any free register, and then Boolean logic instructions are used to set the required values of the program mask bits. The program mask formed is placed in the PSW with the aid of a SPM instruction.

Before return to the externally stored program, the previous value of the program mask stored in the return register must be recovered.

## 5.6. Supervisor Call

The ES EVM system of instructions includes a special instruction by means of which the program can access the supervisor:

SUPERVISOR CALL    0A   SVC   RR    $I_1$

This instruction is of a particular format. No registers participate in this operation, and the second byte of the instruction is a byte of immediate data. It is only this instruction that does cause a particular interrupt called supervisor call. Byte  $I_1$  is placed in positions 24 through 31 of the old PSW, and bits 16 through 23 are packed with zeros. Next, the old PSW is placed into a double word at address  $20_{16}$ , and fetched from a double word at address  $60_{16}$  is the new PSW starting the supervisor block responsible for handling this type of interruptions.

The byte of immediate data from the SVC instruction placed in the old PSW is a coded message on the service needed by the application program. The value of the interrupt cause code for each type of service provided by the control program can be read in the description of a corresponding supervisor.

We shall use only one type of supervisor call. The instruction

SVC   0A   0E

finds its application in the *Disk operating system* (DOS/ES) to tell the supervisor about the end of the application program execution.

## CHAPTER 6

# CHANNEL ORGANIZATION AND INPUT/OUTPUT

### 6.1. Information Interchange Principles

Prior to execution, any program must be loaded into the machine memory (storage). Besides, to perform its functions, the program needs source information which must be entered in the memory. The program processing results turn up in main storage, but in the form unsuitable for further use. The data must be output from the computer memory. This chapter deals with the facilities for information interchange between the main storage of the computer and peripheral devices (external with respect to the CPU).

INPUT (read) is information transfer from a peripheral device to the main storage.

OUTPUT (write) is a reverse process of information transfer from the memory to a peripheral device. To perform input/output operations various *input/output devices* are connected to the computer. These devices are designed for information conversion from the internal representation in the machine memory to the language of an information-carrying medium and vice versa.

According to their functions, the input/output devices that can be attached to a computer are divided into several groups. There are storage devices (magnetic tape, disk, drum units) which extend the main storage, but operate at a far lower speeds, though have far larger storage capacities (tens and hundreds of millions of bytes). There are still other groups of devices for information input from an intermediate medium (punched cards, paper tape) and devices for information output to an intermediate medium, printing devices, and devices for direct communication between the man and computer (an example is a typewriter) and others.

In the first generation computer the input/output devices operated directly under the control of the CPU which converted the information internally represented in storage into the form suitable for the I/O device, gave instructions to carry out operations to the I/O device, etc. The system of instructions of those machines included many I/O instructions different for each device incorporated by the computer configuration.

Such an approach to organization of input/output limited the set of I/O devices which might be connected to the computer, since each I/O device required its own instructions. As a result, a computer could employ only those I/O devices which were taken into account in the computer design. If the I/O devices available did not satisfy the requirements of a user, new instructions should have been included. i.e. in fact a new machine had to be designed.

Modern computers are free from this drawback and allow attachment of any type peripheral devices. This is obtained by dividing the device into an executive part (the very input/output device in the previous sense) and a control part (unit) also known as a *controller*. The I/O device performs all physical operations such as rewinding magnetic tape, printing or punching a character, etc. What character is to be printed or punched, or what is to be recorded on a moving tape is determined by the controller. Regardless of the I/O device type, the controller receives information from the main storage or transfers it to the main storage always in a standard format, and converts the information to the required format by itself. This accounts for the possibility of connecting any type I/O devices to the computer. In other words, the controller is an interfacing device used between the processor and an I/O device.

Another problem arising in attaching an I/O device to a computer consist in different information processing speeds of the processor and the I/O device. Processors of modern machines are capable of information transfer to and from storage at a speed from hundreds of thousands to several millions of bytes per second. On the other hand, the I/O devices always have mechanical moving parts because of which they operate far slower. For instance, a best punched tape reader can transfer data to storage at a speed not above 1500 bytes per second. If a processor has to wait for completion of an input from a paper tape, then the processor idle time will exceed 99 per cent. This is typical of the first generation computers, but cannot be tolerated nowadays, especially in business applications which involve input/output operations.

The design of modern computers allows for concurrent operation of a central processing unit (CPU) and input/output devices. The CPU only starts the input/output operation which is then continued independently, while the processor proceeds on with its work. More than that, the processor is allowed to operate concurrently with execution of several input/output operations.

Since the I/O devices operate independent of the CPU, the information transfer operations performed between the storage and controller of the I/O device must be carried out by special facilities independent of the CPU. These facilities are called *input/output channels* or simply channels. The channel itself is a computer which like the CPU executes a certain program comprised by storage-resident instructions. The difference is that the channel instructions are highly specialized and apply solely to the input/output operations.

The operation of the channel starts on a special command issued for the CPU which then continues its operation. Upon completion of the I/O operation the channel tells the CPU about this producing a special signal of input/output interrupt.

Generally, several I/O devices are attached to a channel, but their services may be different.

High-speed I/O devices, such as magnetic tape, disk or drum storage units are connected to *selector channels*. The selector channel carries out information interchange with the I/O devices attached to it only one device at a time. It is said to operate in *burst mode*.

*Multiplexor channels* are used mainly with relatively low-speed devices, such as a typewriter or teletype, card read/punches, and printers. The operating speed of this channel is quite sufficient to provide simultaneous service to several low-speed I/O devices, interrogating in turn each device and performing information interchange with those devices which are ready to transmit a successive byte. Actually, the multiplexor channel has several *subchannels*, each of which is able to transmit characters from a different device so as to interleave characters from several devices at the same time. The channel is said thus to operate in *multiplex mode* or *byte mode*.

The multiplexor channel may also be used in burst mode with high-speed devices, such as tape, disk, or drum. However, the channel cannot serve them in its conventional multiplex mode, for the operating speed of these devices is too high. The multiplexor channel may start an interchange operation with a high-speed input/output device only when it is not engaged in servicing any other I/O device. No other device has access to the channel until high-speed transmission is complete. As we have said above, in servicing high-speed I/O devices, the multiplexor channel, like the selector channels, operates in burst mode.

Finally, let us consider the operations performed by the CPU concurrently with input/output operations. Suppose, the program is to enter source data from punched cards and the corresponding *channel program* has been loaded. The CPU continues its work, but it turns out that until completion of the entry operation, the program cannot continue the computation for there are no source data. Therefore the CPU must be changed over to a *wait state* until completion of entry. This brings to naught all the efforts to interleave operations.

The answer to this problem lies in *multiprogramming*. This concept is best explained by the following example. Suppose, we have two programs. One of the programs is a computational type which works as follows: a punch card with several numbers is read, then several minutes are taken by computations operation the results of which are punched into another card, after this a punched card is read again, and so on.

The other program reads information from a magnetic tape and after some editing outputs it on a printer. In the first event, the central processor is fully loaded, but the input/output devices have some idling time. The other program fully loads the printer, but the CPU will constantly wait until printing of a successive line is completed.

As a matter of fact, nothing prevents us from placing both programs simultaneously in different areas of storage, and granting one program the processor, while the other program is waiting for completion of printing. Upon completion of printing a current line, the operation of the former program is interrupted for a short period of time to let the latter program again go over to a wait state, after which the processor is again available to the former program, and so on.

It is this organization of computer operation that is questionably called multiprogramming. It should not be confused with the term *multiprogramming* standing for the ability to run several programs concurrently on a computer which has nothing to do with the skill of programming (program development).

Organization of the multiprogramming mode of operation (in the latter sense) is fairly painstaking matter. However, the user should not be concerned with it, for the multiprogramming is provided by the operating systems available. You should not think that only two programs may be in main storage at a time. The *Disk operating system* (DOS/ES) allows execution of three programs at a time, and certain versions of more powerful *Operating System* (OS/ES) handle up to 52 programs which are in turn, according to certain rules, granted the central processor and other resources.

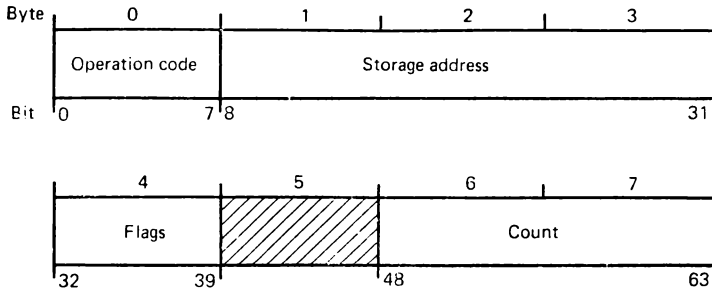
The multiprogramming is most efficient and cost-effective in solving business and other problems with many I/O operations in which case, even with an operating system in use, loading of the CPU in the single program mode is often below 10 to 20 per cent. Therefore, with sufficient number of peripheral devices and channels included in the computer configuration, the multiprogramming can raise the throughput more than five times.

## 6.2. Channel Programs

Thorough knowledge of I/O organization is of importance perhaps only for system programmers charged with writing operating systems and various utilities. The ES EVM operating systems include a set of input/output modules, i.e. standard subprograms carrying out a considerable part of work on information interchange with I/O devices, so that in principle an ordinary programmer may merely write required module calls, having not the faintest idea of actions necessary to execute the operations he has requested for. However, beyond all manner of doubt, well understanding of the organization of I/O operations is of good help to the programmer. This chapter is written with a view to giving the reader a clear idea of organization and abilities of the ES EVM input/output features without overburdening the text with superfluous details.

The ES EVM information input/output operations are performed by the I/O channels according to special *channel programs*. Like

the conventional programs, the channel programs consist of separate commands. The channel program specifies the I/O operation (s) that are to take place and exists as a series of *channel command words* (CCWs) which occupy a double word in storage and have the following format:



The code of the operation (command) carried out by the channel is in the first byte of the CCW. The next three bytes contain the initial address of the field in the main storage, the information interchanges will be performed with. The content of byte 5 in the CCW may be of any value and has no effect on the execution of the command. The last two bytes of the CCW (count) specify the number of main storage bytes used in the I/O operation. After transfer of each byte, the count is decremented by 1, and the storage address is incremented by 1. The execution of the command is terminated when the count becomes equal to zero, if the information on the peripheral data-carrying medium has not been exhausted. Note, that the count and storage address are modified in a special *channel command register*, rather than in the CCW itself. This register is in each subchannel the CCW is read in before execution. The CCW in the main storage remains unchanged.

We now consider in more detail byte 4 of the CCW which contains flags controlling the execution of the channel program.

Bit 32 is the *chain data flag* (CD). If this bit is set to one, then, after the number of bytes specified in the count has been transmitted, a new CCW will be automatically fetched, containing the new storage address, new count, and new flags. However, the operation code of the new CCW is ignored, and the execution of the same operation will be continued. The use of the CD flag makes it possible to write more effective programs, for the information can be at once distributed to the required storage partitions without engaging the central processor for data transfer. The data chain always refers to one physical record on a peripheral medium (a punched card, magnetic tape zone, printing line, etc.) and cannot apply to several physical records.

Bit 33 is the *chain command flag* (CC). If this bit is set to one, then, the current input/output operation has been completed, the next one will be automatically started, i.e. the next CCW with its operation code will be fetched. The command chain is used for processing several physical records and cannot be utilized to process different portions of the same physical record. In any command of the channel program, except the last one, bit 32 or bit 33 must be set to one.

Bit 34 is the *suppress-length-indicator* (SLI). This indicator is utilized when the number of bytes specified by the channel program does not match the number of bytes in the physical record on an external medium, for instance, when less than 80 bytes are read from a punched card. Usually, in this case, the channel reports an incorrect length condition, but setting of the SLI (bit 34) flag in the CCW to one causes the incorrect length indicator to be suppressed.

Bit 35 is the *skip flag* (SKIP). If this bit is set to one, then in read operations, the data from the peripheral device are transferred to the channel, but are not stored in the main storage. These bytes are plainly lost. The read instructions with the SKIP flag may be used to pass over some data on external storage medium or to monitor the information since the bytes being transferred all the same undergo usual checks. The SKIP flag has no effect on work of write instructions.

Bit 36 is the *program-controlled interruption flag* (PCI). The PCI flag instructs the channel to initiate an I/O interruption when a CCW with that bit set to one is encountered. The setting of the PCI bit in a CCW does not affect the execution of the channel programs. An interruption may occur either before, or after the transfer of the first bytes of the CCW data containing the PCI flag. A program-controlled interruption is used to inform the CPU about successful execution of a part of the channel program.

Bits 37, 38 and 39 in the CCW must be set to zero. Nonzero values of those bits are treated as an error in the channel program.

The ES EVM system of channel commands contains six types of commands, the code bit settings of which are as follows.

<i>Command code field</i>	<i>Command</i>
XXXX0000	Invalid command
MMMMMM01	WRITE
MMMMMM10	READ
MMMMMM11	CONTROL
MMMM0100	SENSE
XXXX1000	TRANSFER IN CHANNEL
MMMM1100	READ BACKWARD

The characters X denote the bit position that is ignored (does not affect execution of a channel command). The M characters denote a modifier bit. For example, a LINE PRINT command for a line prin-

ter has the form

000L L001

where the last two bits (01) specify a WRITE instruction, and values of bits LL = 00, 01, 10, or 11 define the number of lines (from 0 to 3) the paper will be moved after printing. Therefore, the LINE PRINT command has four modifications:

01	PRINT, NO SPACING
09	PRINT, SPACE ONE LINE
11	PRINT, SPACE TWO LINES
19	PRINT, SPACE THREE LINES

An operation code containing zeros in four low-order bytes is invalid. When a CCW with such an operation code is fetched, an error in the channel program will be indicated.

The WRITE and READ commands transfer data from main storage to the addressed I/O device and vice versa.

The CONTROL command is a particular case of the WRITE command. This command is used to initiate an I/O operation at an I/O device that does not involve the transfer of data. Examples of a control command are commands for paper skips on a printer, wind of magnetic tape, head positioning on a magnetic disk drives, etc. The count in a control command should not be set to zero, though all the control information is contained by the operation code, and no data are transferred from the main storage. Zero value of the count is treated as an error in the channel program.

The SENSE command is a particular case of the READ command. On this command one or several *sense bytes*, rather than data from I/O medium, are transferred to main storage. These bytes contain detailed characteristic of the addressed device. The SENSE command is used in case of an abnormal termination of an input/output operation, and the supervisor cannot determine, with the data available, what has happened in the addressed device.

The TRANSFER IN CHANNEL (TIC) command leads to execution of the following actions: all fields of the CCW are ignored, except for the storage address which is treated as the address of the next CCW. The TIC command performs an unconditional transfer in the channel program. It cannot be the first CCW in the channel program, and two such commands cannot be executed one after another.

The READ BACKWARD command transfers data from an I/O device to main storage; storage locations are filled in descending order beginning with the data address field in the CCW. The READ BACKWARD command is designed for use with magnetic tape devices that permit the operation to be performed with the tape moving backwards, thus, saving the backward wind time.

The above-described commands are used to develop *channel* programs performing actual input/output operations. A simplest channel program may consist of one command. Sample examples that follow are channel programs for various I/O devices.

**Example.** The operation of information read from punched cards has code  $02_{16}$ . The channel program for reading of a punched card into a main storage area beginning at address 005A60 consists of one command

```
02 005A60 00 00 0050
```

where the count equal  $50_{16} = 80_{10}$  defines the transfer of all 80 card columns.

**Example.** The program that follows, also used for card readers consists of several commands:

```
02 0074A0 80 00 0014
00 000000 90 00 0038
00 0074B4 00 00 0004
```

The first CCW, in which a CD chain data flag is specified, will cause data transfer to main storage from  $14_{16} = 20_{10}$  columns of the punched card, beginning with address 0074A0. The next  $38_{16} = 56_{10}$  columns of the card will be skipped, since in addition to the CD flag, a SKIP flag is specified in the second CCW. The operation code of this CCW is ignored, since it is in the data chain. No data are transferred to main storage, and any storage address, say zero one, may be specified in the second CCW. The last four bytes will be arranged starting with address 0074B4, forming a continuous field with previously stored 20 bytes.

**Example.** The program prepared for a magnetic tape unit is as follows:

```
01 001000 40 00 0200
0C 000000 18 00 0200
```

The program records a zone on a magnetic tape (operation code 01), and then reads it backwards (operation code 0C). Recorded is a zone,  $200_{16} = 512_{10}$  bytes long, and the information is taken from main storage, starting with address 001000. A chain command flag (CC) is set in the first CCW, for which reason, the next channel command will be automatically fetched upon completion of the recording. The second CCW contains SKIP and PCI flags. The information will not be transferred to main storage, recording quality will be checked. After the second CCW has been fetched, a program-con-

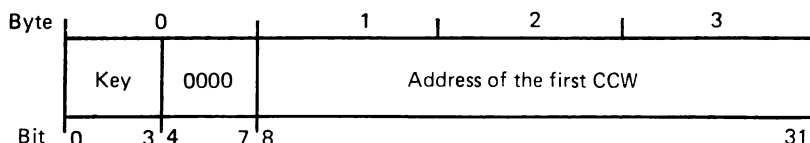
trolled interruption will occur to inform the central processor about completion of recording.

The development of these channel programs is not difficult though it requires that the codes of operations carried out by different I/O devices be well known. Simple I/O devices, such as card and paper tape read/punch units, printers, magnetic tape units, etc. have a limited set of operations and employ relatively simple channel programs. Complicated devices, such as disk drives, have a well developed set of operations comprised by several tens of commands, with the modifications taken into account, and need fairly complicated channel programs.

### 6.3. Channel-CPU Interaction

Overlapping of CPU operation and execution of I/O operations leads to necessity of organizing communication between the CPU and channels. Each time the program is to execute an I/O operation, the central processor must interrogate the channel to see whether it is serviceable and not engaged by another I/O operation. If so, the CPU informs the channel about the location of the required channel program in main storage, and the channel starts the execution of its program by itself. At the same time the CPU continues computations. Upon completion of the I/O operation, the channel must tell this to the CPU which is accomplished by means of input/output interruptions. To transfer the control information between the CPU and channels use is made of two storage locations with fixed addresses.

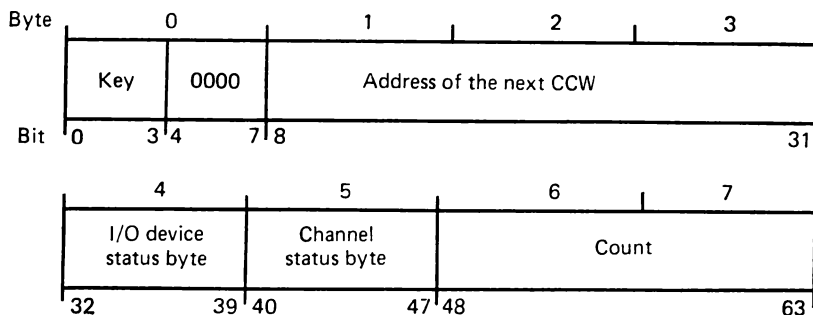
The first location contains a *channel address word* (CAW)—a full word located at address 48<sub>16</sub>. The CAW format is as follows:



The first nibble of the CAW contains a storage protection key<sup>12\*</sup> having the same value as the protection key in the PSW. The following four bits should be set to zero. The three low-order bytes of the channel address word are used for the address of the first CCW of the channel program. The CAW contains all the information the channel needs to start its working. Before addressing the channel, the required value must be placed in the CAW.

Stored in the other fixed location is a *channel status word* (CSW). The CSW information is utilized by the supervisor block handling input/output interruptions to monitor execution of the input/output operations.

The CSW occupies a double word at address  $40_{16}$  and has the following format:



The channel status word is stored in cases when the supervisor may need the information contained in it. For instance, at the end of any input/output operation, when an error is detected in the channel program, in case of hardware faults or failures leading to an abort of input/output, etc.

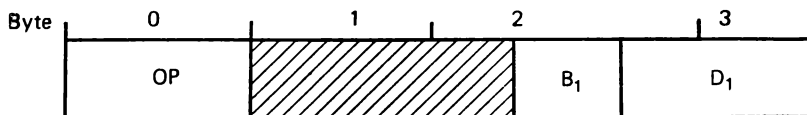
The left-hand byte containing the storage protection key is transferred to the CSW from the CAW which was used in starting the channel program. The next three bytes contain the address of the CCW to which the execution of the channel program has progressed. Usually, this is the address of the CCW which was executed last plus eight, but under certain circumstances stored is the address of the last CCW plus 16. Bytes 4 and 5 of the CSW contain information about the status of the I/O device and channel (their structure in detail will be given later). The last two bytes of the CSW are used to store the value the count has progressed to in the channel command executed last. If the CSW is stored at the end of an operation, the count field is usually set to zero. Otherwise, if the execution of the channel program has been stopped for some reason, the address of the last byte being transferred can be determined against the storage address and initial value of the count in the first CCW and final value of the count in the CSW.

To call the input/output facilities the set of instructions includes four I/O instructions:

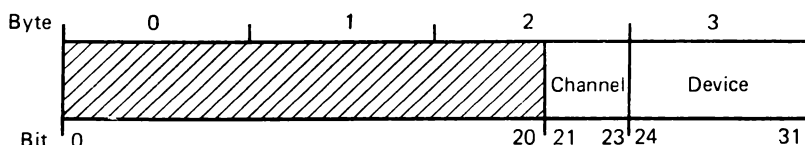
START INPUT/OUTPUT	9C SIO SI
TEST INPUT/OUTPUT	9D TIO SI
HALT INPUT/OUTPUT	9E HIO SI
TEST CHANNEL	9F TCH SI

All the input/output instructions are privileged and can be executed by the CPU only in a supervisor state. They have the SI (storage

immediate) type, field  $I_2$  in them is not used and may be filled with any information:



The effective address of the first operand  $E_1 = (B_1) + D_1$  in the input/output instructions is not a storage address, and is used to define the I/O device addressed by the instruction:



The low-order eleven bits of the effective address specify I/O devices, the very address of the I/O device being specified by the right-hand byte, while the other three bits contain the number of the channel the device is connected to. The high-order bits of the effective address have no effect on execution of input/output instructions.

The input/output instructions are described in detail in Sect. 6.6. Here, only the common principles of their operation are set forth.

Execution of any channel program starts on an SIO instruction. Before this, a record should be made in the CAW containing the storage protection key and address of the first command of the channel program. On the SIO instruction tests are made to see whether the channel and I/O device are serviceable and not engaged with another input/output operation. Next, the CAW and first CCW are checked to see whether they are valid, and, if everything is OK, the channel program is started. The SIO instruction sets a condition code of zero, if the channel program has been started. Otherwise, the condition code gains other values. The SIO instruction generally is followed in the program by a BRANCH ON CONDITION (BC) instruction which passes control to the block handling unusual input/output situations at a condition code other than 0.

The TIO instruction is associated with handling I/O interruptions. It is considered below. Note here that this instruction can be used to check the execution of an input/output operation without interrupt.

The HIO instruction is used to terminate the execution of a previously started I/O operation. Although seldom, this is necessary, for example, when a typewriter is busy with entry of information

from the operator, while the supervisor needs to print an urgent message.

The TCH instruction is used to determine what work is in progress on the channel at a current moment of time. This instruction does not affect the operation at all.

Unlike the other input/output instructions, the TCH instruction analyzes only the channel address. The I/O device address is ignored by this instruction.

The input/output instructions are used to organize communication between the CPU and channels. The channel feedback to the CPU is accomplished by producing signals of I/O interrupts. For the input/output interruptions in detail, see Sect. 6.5. Here we shall consider the organization of their handling by the supervisor. As a result of an input/output interruption, control is passed to the corresponding block of the supervisor and simultaneously is stored in the CSW. Using the interruption code, the supervisor determines what device has caused it, and after an analysis of the CSW, the supervisor finds the cause of the interruption and takes appropriate measures.

There is nothing extraordinary here, except for the following circumstances. Information input/output on the ES EVM is organized so that several I/O devices may operate concurrently. If that is the case, it is possible that one of these devices causes an input/output interruption exactly at the instance of time when the supervisor is engaged in handling another I/O interrupt. The after-effects of the latter input/output interrupt would be disastrous. In the course of any input/output interruption the old PSW and CSW are stored in the same storage locations, for which reason, all the information related to the former interrupt would be completely ruined.

For these reasons, means are required to disable input/output interruptions. In the ES EVM, use is made of a system mask contained in bits 0 through 7 of any PSW. Bits 0 through 6 are used for interrupt enable-disable on channels 0 through 6 respectively. When the system mask bit is set to one, the corresponding channel is allowed to cause interrupts. If this bit is zero, the channel is not allowed to enable interrupts. If that is the case, the channel is said to be *masked off*. Bit 7 of the PSW is intended for masking external interrupts which also can occur at any time instant and owing to various causes.

Masking input/output interruptions differs from masking program interruptions. If a program interruption is masked, the interruption signal merely disappears. An input/output interruption signal must not disappear, for any interruption of this kind needs to be processed. Therefore, the signal of a masked I/O interrupt is stored nonprocessed in the channel, subchannel, or I/O device controller (the place of storage is dependent on the type of I/O device and cause of interruption). This signal will cause an interrupt as soon as a PSW in

which the corresponding system mask bit is set to one is loaded as the current PSW. After an interrupt has occurred, the interruption signal is cleared and not stored at all. However, till this time instant the subchannel cannot execute new input/output operations.

There are two tools in the ES EVM which can be used to control input/output interrupts.

If a TIO instruction interrogates an I/O device storing the non-processed interrupt, the corresponding CSW is stored and the interruption signal is cleared. Thus, by means of a TIO instruction you may organize interruption handling without executing interruptions as such.

To make masking input/output interruptions easier, the CPU set of instructions includes a special instruction:

SET SYSTEM MASK      80   SSM   SI    $I_2B_1D_1$

The second operand ( $I_2$ ) is not used and does not affect the execution of the instruction. One byte of data from storage defined by the address of the first operand replaces the system mask in the current PSW. The other part of the PSW remains unchanged. The SSM instruction is privileged and can be executed by the processor only in a supervisor state.

Now we can formulate the general procedure of input/output programming on the ES EVM.

1. All the input/output interruptions are masked.
2. The protection key and address of the first CCW of the channel program are placed in the CAW.
3. An SIO instruction is issued which produces a condition code set to zero. Otherwise, control is passed to the block handling special input/output conditions.
4. Input/output interruptions are allowed. Those must be masked before issuing an SIO instruction, for this instruction stores the CSW in certain instances.
5. While the channel executes its program, the CPU continues computations.
6. When the CPU program reaches the point at which it cannot be executed further until the input/output operation is terminated, a check is made to see whether the input/output is completed. If so, the program continues its work. If not so, the processor is changed over to a *wait state* in the single program mode, or becomes available to another program in the multiprogramming mode.
7. Upon completion of the execution of the channel program an interrupt occurs. All the channels in the new PSW loaded as a result of the interrupt must be masked off. The input/output interruption handling program checks to see whether the operation is properly terminated, and if an error is detected performs the corresponding actions.

8. Upon completion of handling an interruption, the channels are unmasked. In practice, a user's program requests the supervisor to execute an input/output operation and all the above-listed actions are carried out by the supervisor, so that each programmer need not write his own program to control input/output operations. Nevertheless, correct and efficient use of the input/output capabilities offered by the supervisor dictates that the programmer be well conversant with how the channels, CPU, supervisor and user's program work and interact.

#### 6.4. Status Bytes

The input/output control principles considered in the previous section require presence of information on the status of the channels and I/O devices to make decisions on actions to take. This information is contained in *status bytes* found in each subchannel and each I/O device. One status byte of the channel and one status byte of the I/O device are stored in the CSW in response to an I/O interruption, and also in certain other cases. If the information contained in the CSW is insufficient to determine what has happened in the device, the supervisor may request data in more detail stored in *sense bytes* contained in each I/O device. This request is executed on a channel command SENSE.

The structure of status bytes is so selected that in usual execution of input/output operations all their bits are set to zero. Ones in one or several bit positions of status bytes indicate an erroneous or tolerable, but particular input/output condition.

We shall designate the channel status byte with the character S and use a subscript in reference to separate bits of this byte.

$S_0$  is *program-controlled interruption* (PCI). Bit  $S_0$  is set to one by any CCW which contains one in bit position 36 (PCI flag), and an interruption signal is produced in the subchannel. If the CCW containing the PCI flag simultaneously initiates bit  $S_2$  indicating an error in the channel program,  $S_0$  remains set to zero.

$S_1$  is *incorrect length*. Bit  $S_1$  is set to one, if no SLI flag is specified in the CCW and one of the following two events has occurred: (1) the physical end of a record is reached, while the count has not yet counted to zero, or a data chain is specified; (2) the count has counted to zero, and no data chain has been specified, but the physical end of the record is not reached. With the  $S_1$  set to one, the execution of the channel program is terminated and an interruption signal is produced.

$S_2$  is *channel program check*. Bit  $S_2$  is initiated when an error is found in the CAW or CCW format, i.e. in the following cases:

Bits 4 through 7 do not contain zeros.

The address of the first CCW in the CAW is not a multiple of eight, or exceeds the storage capacity of the machine.

The first CCW in the program is a TRANSFER IN CHANNEL (TIC) instruction.

Two TIC instructions are executed one after another.

In the TIC instruction the address of the next CCW is not a multiple of eight, or exceeds the storage capacity of the machine.

A count field in any CCW other than TIC is zero.

Bits 37 through 39 in any CCW other than TIC are nonzero.

There is an illegal op(eration) code (zeros in bits 4 through 7) in any CCW not included in the data chain.

Note, that bit  $S_2$  is not initiated, if the CCW has a correct format, but cannot be executed by a device, for instance, if a READ instruction is issued to a printer. If  $S_2 = 1$ , the execution of the channel program is terminated, and an interruption signal is generated.

$S_3$  is *storage protection check*. Bit  $S_3$  is set to one, if the protection key in the CAW does not match the key of the storage location involved in the information interchange. If that is the case, the protected byte of the storage is not accessed, execution of the channel program is terminated, and an interruption signal is generated.

$S_4$  is *channel data check*. Bit  $S_4$  is initiated, if a character of improper parity is transferred between the channel and I/O device. The data transfer may continue, but the data chain is always broken, an interruption signal is issued. The channel is forced to form the correct parity for the bytes passing through it, for which reason, testing the value of bit  $S_4$  is sometimes the only method of detecting an error.

$S_5$  is *channel control check*. Bit  $S_5$  is initiated, if a fault occurs in the channel equipment. The condition of the channel is unpredictable, and the contents of the other fields of the CSW may be invalid.

$S_6$  is *interface check*. The input/output interface is a system of links, buses, signals and algorithms of input/output providing attachment or various I/O devices to a channel and control of all input/output operations. Bit  $S_6$  is initiated when an invalid signal occurs on the I/O interface. It usually indicates malfunctioning of an I/O device. An example is a device improperly responding to a channel signal. The current operation is immediately concluded and an interruption signal is issued. As a rule, one in bit positions  $S_4$ ,  $S_5$  or  $S_6$  indicates malfunctioning of the equipment. In this case, the machine must be given a repair or adjustment service.

$S_7$  is *chaining check*. Bit  $S_7$  is initiated if during data chaining channel overrun occurs, because the I/O data rate is too high for the particular resolution of data addresses, i.e. the channel has insufficient time (to some reason) to decode a next in turn CCW in the chain within the specified time interval, namely before a successive byte arrives. The operation is terminated and an interruption signal is issued.  $S_7 = 1$  usually indicates that too many I/O operations are being exe-

cuted concurrently and either the channel, or main storage is over-run.

The eight bits of the I/O device condition byte which will be denoted by the character  $U$  have the following meanings:

$U_0$  is *attention!* Bit  $U_0$  goes over to a one state, if the operator has depressed a special button on the I/O device. As this happens, an I/O interruption signal is produced which is not associated with any channel programs. Such an interruption is a standard means to draw attention of the supervisor to an event in the device.

$U_1$  is *status modifier*. This bit has no its own function and is used only together with bits  $U_3$  and  $U_5$  to modify their usual value.

$U_2$  is *control unit end*. Bit  $U_2$  is set to one, if the control unit (controller) has completed its part of an I/O operation, i.e. the information transfer is completed, all control signals are issued, and the I/O device has to carry out mechanical actions, say, to stop magnetic tape motion, or pull paper on a printer. This bit is used only when the control unit has been interrogated and is in the busy state.

$U_3$  is *busy*. Bit  $U_3$  remains initiated all time while the I/O device is engaged in execution of an I/O operation. If  $U_1$  is initiated simultaneously, this indicates that the control unit has been interrogated and is busy.

$U_4$  is *channel end*. Bit  $U_4$  is set to one and this condition denotes that the data transfer portion of the operation is complete and channel facilities are no longer needed.

$U_5$  is *device end*. Bit  $U_5$  is initiated when the I/O device terminates an I/O operation and can execute the next I/O operation, and also when a previously inoperational device goes over to the 'Ready' state. If concurrently with  $U_5$  status modifier  $U_1$  becomes set to one, this indicates an abnormal but tolerable end of an operation.

$U_6$  is *unit check*. Bit  $U_6$  is initiated, if some bits are initiated in sense bytes of the I/O device. To determine what has happened, the program should check these bytes.

$U_7$  is *unit exception*. Bit  $U_7$  is initiated, if a condition is detected that usually does not occur, but is tolerable. Examples are end of card deck, or end of magnetic tape.

For the last CCW in a string of instructions, initiation of signals *Channel end* and *Device end* indicates completion of the channel program, and in this case an input/output interruption occurs. If the case is that the bit of instruction chaining in the CCW is set to one, then, after these signals have appeared, the channel checks to see whether the last I/O operation is executed correctly. If all bits (except, may be,  $S_0$ ) in the channel status byte are zeros, and all bits, except  $U_4$  and  $U_5$  (and, perhaps,  $U_1$ ) in the I/O device status byte are set to zero too, the previous operation is normally terminated and the next CCW is automatically fetched. The address of the next CCW is commonly equal to the address of the previous CCW plus eight,

but if at the same time the bits *Device end* and *Status modifier* ( $U_5$  and  $U_1$ ) are initiated, the address of the CCW is increased for 16, i.e. one channel command is omitted. The instruction that is usually omitted is a TIC command which passes control to one of the channel program branches upon normal termination of the operation which is followed by another branch which corresponds to an abnormal termination of the operation. This is the way, the branches and loops in channel programs can be organized which is widely used in preparing channel programs for magnetic disk and other direct-access units.

The number of sense bytes and value of certain bits in them are dependent upon the type of the input/output device. However, the values of the first six bits are similar for all I/O devices.

Bit 0 is *command reject*. A channel command has been received which the I/O device is not designed to execute (invalid op-code).

Bit 1 is *intervention required*. This bit indicates conditions such as an empty hopper in a card punch or the printer being out of paper, etc.

Bit 2 is *bus out check*. The device has received a data byte or a command code with an invalid parity from a subchannel.

Bit 3 is *equipment check*. This bit indicates that the device or the control unit has detected equipment malfunctioning, such as an invalid card hole punch or printer buffer parity error.

Bit 4 is *data check*. Setting this bit to one identifies errors associated with the recording medium. An example is an invalid card code.

Bit 5 is *overrun*. The channel has failed to respond in time to a request for service from the device with resultant loss of information. This generally means that too many input/output operations are being carried out at a time.

The channel and I/O device status bytes are reset during their recording in the CSW, and the sense bytes are reset at the beginning of each read or write operation. Thus, data on the input/output system status are not transferred to storage twice. There are two exceptions to this rule:

— Bit  $U_3$  (*busy*) remains turned on all the time of I/O device operation;

— Bit  $U_1$  (*status modifier*) and unit check ( $U_6$ ) produced by it remain set to one until their cause is eliminated.

### 6.5. Input/Output Interruptions

Input/output interruptions provide a means for the CPU to change its state in response to conditions that occur in I/O devices or channel. When an input/output interruption occurs, the current PSW is stored at address  $38_{16}$ , the interruption code containing the address of the channel or I/O device in the low-order eleven

bits, and zeros in the high-order five bits. The new PSW pertaining to the interruption handling program is loaded from the double word at address  $78_{16}$ . In addition to the PSW, the CSW consisting of four fields: CAW, channel and device status bytes (U and S), and count, is stored at location (address)  $40_{16}$  during an input/output interruption.

Input/output interruptions occur in the following four cases:

1. *Work end.* This interruption indicates normal or abnormal termination of the channel program. At least *Channel end* ( $U_4$ ) must be initiated in the unit status bytes. The full CSW is stored:

CAW   U   S   count

2. *Program-controlled interruption.* This occurs when bit  $S_0$  is set to one. The execution of the channel program is continued for which reason the unit status byte is not stored. The channel status word contains the following fields:

CAW   0   S   count

3. *I/O device interruption.* This takes place, if an event has occurred in a device, the supervisor must respond to. An interruption signal of this type is produced, for instance, when an I/O device not ready is turned on, or when the operator depresses the ATTENTION button on the device. The I/O device interruption is associated with no one of channel programs, and solely the status bytes are stored:

0   U   S   0

4. *Interface check.* This interrupt occurs only in a multiplexor channel, and since it is associated neither with any of the channel programs, nor actual I/O devices, only the channel status byte is stored:

0   0   S   0

In the channel status byte at least bit  $S_6$  is set to one (an interface check).

The signals of work end interruptions and program-controlled interruptions are produced in a subchannel. I/O device interruptions occur in a device or control unit, and interface check interruptions, directly in the channel. An interruption signal from the I/O device or subchannel enters the channel as soon as it is free and can receive it. If the channel is masked, the interruption is pended until it is cleared by a TIO (test I/O) instruction or unmasked. I/O device interruptions not yet transferred to a subchannel are also cleared by a SIO instruction.

### 6.6. Input/Output Instructions

The operating principles of the four input/output instructions are described in Sect. 6.3. This section (optional for the first reading) deals with the algorithms for their execution.

9C	SIO	START INPUT/OUTPUT
----	-----	--------------------

The purpose of the SIO instruction is to start the channel program, but after many tests to see whether the input/output operation can be executed. If the instruction has started input/output, the condition code is set to zero. Otherwise the condition code is set to 1, 2 or 3 depending on the cause which interferes with the execution of the operation. When the condition code is one, the status bytes of the I/O device and channel (U and S) are additionally stored, while the other fields of the CSW remain unchanged.

Like the other input/output instructions, the SIO instruction is executed in steps. If the requirement being tested is satisfied, the condition code (CC), is set, and the operation is stopped, otherwise the next step is executed. The operation of the SIO instruction comprises the following steps.

1. If the channel is not operational, set the CC to 3.
2. If the channel is off-time, set the CC to 2.
3. If a nonexistent subchannel is specified in the multiplexor channel, the CC must be set to 3.
4. If a pending interruption is stored in the subchannel, set the CC to 2.
5. If the addressed subchannel of the multiplexor channel is engaged in an input/output instruction, set the CC to 2.
6. If there are errors in the format of the CAW or the first CCW, set  $S_2$  to 1, CC to 1, and store  $CSW = 0$ , S.
7. Start the channel program.
8. If a PCI condition is specified in the first CCW, set  $S_0$  to 1. This initiates an interruption signal in the subchannel, but the channel program execution is continued in a usual way.
9. In case of an interface check, set  $S_6$  to 1, CC to 1, and store  $CSW = 0$ , S.
10. If the device is inoperational, or its reply to a signal from the channel is invalid, set CC to 3.
11. If the control unit is engaged in another input/output operation, set  $U_1$  to 1,  $U_3$  to 1 (conditions 'Status Modifier' and 'Busy'), CC to 1, and store  $CSW = U$ , S.
12. If  $U = 0$ , set CC to 0 (the channel program execution is started).

13. If specified in the first CCW are instruction chains and  $U = 08, 0C, \text{ or } 4C$  (bits  $U_4$ , and  $U_5$ , perhaps  $U_1$  are set to 1), set CC to 0. This means that the execution of the first channel command is completed, and the execution of the channel program is continued.

14. Set the CC to 1 and store the  $CSW = U, S$ . If  $U = 08, 0C, \text{ or } 4C$ , the execution of the channel program has been concluded. Otherwise, the operation is not executed.

Therefore, the condition code after the SIO instruction is set to 3, if the channel, subchannel, or I/O device are inoperational. The  $CC = 2$  means that the channel, or subchannel has not yet completed the previous I/O operation. The  $CC = 1$  indicates that the channel program has been completed, or not executed at all. To clarify the situation, analyze the channel and device status bytes. With the  $CC = 0$ , the execution of the channel program has been started.

9D	TIO	TEST INPUT-OUTPUT
----	-----	-------------------

The state of the addressed channel, subchannel, and device can be determined by means of the TIO instruction. Like in the other input/output instructions, during the execution of the TIO instruction, checks are made in sequence of the conditions and a condition code is set.

1. If the channel is inoperational, set the CC to 3.
2. If the channel operates in the off-line mode, set the CC to 2.
3. If a nonexistent subchannel is specified in the multiplexor channel, set the CC to 3.
4. If a pending interruption is stored in the subchannel for the addressed I/O device, set the CC to 1, store the  $CSW = CAW, U, S$ , Counter, and clear the interruption signal.
5. If a pending interruption is stored in the subchannel for another I/O device, set the CC to 2.
6. If the I/O device is inoperational, set the CC to 3.
7. If the control unit is busy, set  $U_1$  to 1,  $U_3$  to 1 (conditions 'Status modifier' and 'Busy'), CC to 1, and store  $CSW = 0, U, S, 0$ .
8. In case of an interface check, set  $S_8$  to 1, CC to 1, and store  $CSW = 0, 0, S, 0$ .
9. If the unit status byte (U) is not equal to zero, set the CC to 1, and store  $CSW = 0, U, S, 0$ .
10. Set the CC to 0.

Therefore, the condition code is set to 3, if the channel, subchannel, or I/O device are inoperational. The condition code is set to 2, if the subchannel has not yet completed another input/output operation.  $CC = 1$  means that some bits in the channel and device status bytes are other than zero. With the condition code set to zero, the SIO instruction can be executed by the channel.

9E	HIO	HALT INPUT-OUTPUT
----	-----	-------------------

The execution of the HIO instruction consists of the following steps:

1. If the channel is inoperational, set the CC to 3.
2. If the channel operates in the off-line mode, set the CC to 2.
3. If a nonexistent subchannel is specified in the multiplexor, set the CC to 3.
4. If a pending interruption is stored in the subchannel, set the CC to 0.
5. In case of an interface check, set  $S_6$  to 1, CC to 1, and store  $CSW = 0, S$ .
6. If the I/O device is inoperational, set the CC to 3.
7. If the control unit of an I/O device is busy, set  $U_1$  to 1,  $U_3$  to 1, CC to 1, and store  $CSW = U, S$ .
8. Set the CC to 1 and  $CSW = 0, 0$ .

The condition code is set to 3 under the same conditions as in the previous input/output instructions. With the CC = 2, data transfer in the off-line mode is at once stopped, while the device operates till the conventional termination. With the CC = 1, the data transfer is stopped, and the channel issues an instruction to halt, as soon as the I/O device requests the next in turn byte of data. Zero value of the condition code means that the operation has been terminated, and nothing can be brought to halt.

9E	TCH	TEST CHANNEL
----	-----	--------------

In execution of the TCH instruction only the channel address is dealt with. Neither subchannels, nor I/O device participate in this operation. The TCH instruction sets the condition code as follows.

1. If the channel is inoperational, set the CC to 3.
2. If the channel operates in the off-line mode, set the CC to 2.
3. If a pending interruption is stored in the channel, set the CC to 1.
4. In other cases, set the CC to 0.

### 6.7. Initial Program Loading (IPL)

When a computer is started there are no instructions in main storage at all, and to load a program into main storage additional facilities are required. To this end, the ES EVM console has load-unit-address switches which are pressed to select the address of the channel and I/O device, and a special initial load key (the IPL key). Pressing this key causes the following actions.

1. All status bytes and sense bytes are reset and all operating devices are stopped.

2. The processor executes an SIO instruction using the address of the channel and device selected on the console. As this happens, the CAW and the first CCW of the channel program have the following values:

CAW = 00 000000

CCW = 02 000000 60 00 0018

The channel command having op-code 02 is a standard READ command which can be executed by all the input devices. In the CCW the chain-command and suppress-incorrect-length-indication flags are on, for which reason, after read of  $18_{16} = 24_{10}$  bytes into the first 24 bytes of storage, the next CCW will be fetched at location 8.

3. Upon completion of the channel program, the central processor will start its operation, the double word at location 0 being fetched as the PSW.

Therefore, the IPL key is used to read the PSW and two CCW into storage. The channel commands read are then used to load the remainder of the program, after which the read PSW is loaded from location 0, and the CPU starts its operation.

Upon completion of the IPL procedure no input/output interruption occurs, for which reason the channel program execution monitoring is accomplished by the operator with the aid of a special IPL lamp found on the computer control panel. This lamp lights when the IPL key is pressed, and goes dark after normal termination of the channel program and successful loading of the PSW.

## **CHAPTER 7**

### **INTRODUCTION TO ASSEMBLER**

#### **7.1. Symbolic Programming**

The procedure for writing any program consists of the following steps:

- Problem statement;
- Solving method selection;
- Algorithm development;
- Translation to a flow-chart;
- Writing source code.

The first four steps involve creative work which requires knowledge, experience and often inventiveness. Coding a program, i.e.

writing a set of hexadecimal digits depicting instructions and data is a very tedious error-prone procedure. Most people are incapable of carrying out such work for a long period of time. The result is many errors in programs to detect and correct which often takes more time than all the previous work.

Programming in the machine code is especially hindered by the following circumstances:

- It is difficult for most people to remember numerical codes used to represent instructions and data and is much easier to memorize mnemonics;

- Representation of various constants in the hexadecimal form is a very tedious procedure;

- Each element of a program must be assigned an address;

- Correction of errors and program modifications are rather complicated since it often involves changes to many addresses in instructions.

The authors are quite certain that the readers who thoroughly carried out the exercises given in the previous chapters have felt this.

With gaining programming experience, it became evident that the amount of program coding work is materially reduced, if *symbolic designations* of program elements are substituted for actual addresses in the instructions. Use of a symbolic language makes the program writing much easier, since it is more convenient to write instructions in an alphabetical form than in numerical codes.

As an example, we shall consider a segment of a program performing a computation  $X = \min(A, B)$  where all variables are fixed-point binary numbers.

<i>Machine codes</i>				<i>Symbolic addresses</i>	
004000	58	2	0 9 030	L	2,A
004004	59	2	0 9 034	C	2,B
004008	47	4	0 9 010	BC	4,L1
00400C	58	2	0 9 034	L	2,B
004010	50	2	0 9 038	L1	ST
					2,X

It is evident that the program in the right-hand column is much easier to remember than the same program written in the machine language. More than that, with a symbolic program you do not need to write addresses of instructions which are not addressed. Therefore, in the above example label L1 is used to identify only the ST instruction to which a branch will be made on a BC instruction.

However, a symbolic program cannot be directly executed by a computer, for the computer accepts only instructions represented in hexadecimal form. To this end, after the program has been written, corresponding digital codes should be substituted in each instruction for symbolic designations. Conversion of a program from a symbolic language into a machine language is called *translation*. The tran-

lation rules are fairly simple, and from the mid of 50s this monotonous and intensive work is carried out by the computer itself. Programs are designed which are called *translators* to translate source statements (symbolic instructions) into the machine language for a specific computer. The input to the translator is a program written in a symbolic language, and the information output from it is a program in the machine language. The latter program must be additionally processed by a special program known as an *Editor* which supplements it with subprograms and arranges in an actual area of main storage where it will be processed. The program obtained as a result of the Editor work can be directly executed on a computer.

The symbolic programming language for the ES EVM is called the *Assembler language*.

A program written in the Assembler language is also known as a *source module*, the result of the translator (compiler) work is called an *object module* while the result of the Editor work, an *absolute module*.

As mentioned above, the instructions comprising the symbolic or high-level language (the source module) must be translated to machine language, before the computer can execute these instructions (the object module). We call this process of translation *assembling* or *compiling* the program. We usually associate the term *assembly* with the translation of a symbolic language to machine language, and the term *compiling* or *compilation* with the translation of high-level languages to machine languages. Translators that convert only a portion of the source statements at a time into machine-executable code are called incremental compilers or *interpreters*.

During the operation of the translator (compiler) we may obtain listings of the source and object modules. This output is called the *program listing*.

Advantages of the Assembler language over machine languages are as follows:

- Symbolic designation of operation codes (op-codes);
- Symbolic addresses of program elements;
- Possibility of writing constants in the form convenient for a person;
- Automatic boundary alignment.

Besides, the abilities of the Assembler language include *macros* which allow frequently encountered sequences of instructions to be written in a condensed form by one special symbolic statement known as a *macro instruction*.

## 7.2. Writing Symbolic Statements

Symbolic statements comprising Assembler language programs are generally written on *coding* or *program sheets* which are special coding forms. Each *line* of a coding sheet will be then keypunched into one

punched card. The coding sheet has space for 80 columns of information forming positions for characters, one position for a character.

Each line of the coding sheet consists of two fields: a *statement field* (columns 1-71) used for actual coding of each Assembler statement, and an *identification field* (columns 73-80) used to identify a program and to indicate the sequence of cards as well.

Column 72 is known as a *statement continuation column*. If an instruction requires more than one line of coding, it may be continued on the next line. In such cases, a nonblank character is coded in column 72 of the line to be continued. That is, if two lines are required for an instruction, only the first one will contain a character in column 72. To continue an assembler language statement onto the next line, simply put any nonblank character in column 72 and continue the statement starting at column 16 of the next line, which is termed the continuation column. The positions to the left of this column should contain blanks. Only one line of continuation is permitted. Exceptions are macro instructions which may have more than one continuation line.

The statement field in turn consists of four fields (from left to right), the name, operation, operands, and comments field in that order. The coding sheet is designed so that these entries for each instruction are always placed in the same columns in a uniform manner.

The *name field* serves the purpose of placing the *symbolic name* (*label*) which may be used by the programmers for referencing the statement. A symbolic name consists of from one to eight characters, the first of which must be a letter in column 1 of the form. There should be no blanks within a symbolic name.

The name field is optional. It is generally filled only for those program statements which are referenced in other statements.

The name field, if used, is followed by at least one blank space. Then comes the *operation field*. If no name field is used, the operation field must begin to the right of the first column of the coding sheet (form).

To make reading a program (sample) listing more convenient, it is good practice to start the operation field in position ten of the operation column (sheet columns 10-14), though it is optional. The operation field must be present in any symbolic statement. This entry is used to specify the actual operation to be performed in a mnemonic form. The mnemonics must be written exactly as prescribed, with no embedded blanks. There must be at least one blank separating the operation field from the next field in the statement, the operands field.

The *operands field* identifies and describes the data fields to be operated on. Depending upon the instruction format, they may be one or two operands associated with each instruction. Most instructi-

ons utilize two operands, separated by a comma (no space after the comma).

Certain symbolic statements may contain no operands field at all.

The *comment field* is used to supply explanatory or descriptive information about a particular instruction. The comment appears in the operand section of the coding sheet and is separated from the last operand by at least one space. Note that the comment entry is strictly optional and does not affect the execution of the program in any way. It may, however, be enormously helpful to the programmer by providing brief reminders of the purpose of specific instructions. Comments are with no changes transferred into the sample (program) listing.

The comment included along with an instruction may not go beyond column 71 of any line (beyond the continuation column). In the statements containing no operands fields, the comments are separated from the operation field by a comma which is preceded and followed by at least one blank of space.

If more comprehensive comments are desired, certain statements of the Assembler language may contain a special *comment statement* in which whole lines are reserved for comments only. By coding \* (an asterisk) in column 1, the entire line, from columns 1-71, is treated as a comment and may contain any information you like.

### 7.3. Assembler Alphabet. Terms and Expressions

The alphabet of the Assembler language includes English letters from A to Z, special characters  $\text{\textcircled{X}}$ ,  $\text{\text{#}}$  and  $\text{\textcircled{A}}$  which are used as letters (sometimes the dollar sign \$ is used in place of  $\text{\textcircled{X}}$ ), digits from 0 to 9, special characters + - \* / = ( ) . , ' , & and blank.

In addition any character of the EBCDIC (DKOI) code may be used in the comments and symbolic lines.

In the Assembler language any operand is written as an *expression* which after assembly receives a certain value. Such an expression may be a single *term*, or a combination of *several terms*, arithmetic operation signs, and parentheses.

The Assembler language knows five types of terms each of which is considered in detail below:

- Symbolic name;
- Value to the location counter;
- Self-defining term;
- Literal;
- Reference to the length attribute.

Defined for each term are its *value* and *relocatability attribute*. The term value can be assigned during the assembly operation, or may be defined by the term itself (for the self-defining terms). The relocatability attribute indicates whether the value of a term will

change due to relocating the program to another area of storage. The term is called *absolute* if its value does not change because of relocating the program. Examples of absolute terms are a register number in a machine instruction or an immediate operand in instructions of the SI format. A term is *relocatable*, if its value changes in relocating the program. An example of a relocatable term is the address of some program element.

The *symbolic names (labels)* are used for referencing certain instructions or storage areas. The labels may contain from one to eight characters first of which must be a letter. The other characters may be letters and digits. No blanks and other special characters are permitted within the labels.

Examples of valid symbolic names are as follows:

```
RUSSIAN
GAMMA
X
AGENT007
@ 43 //
IJEYZWZZ
```

The following examples are invalid symbolic names:

IJEYZWZZ	(more than eight characters)
7XYZ	(the first character is other than letter)
G.FORD	(contains a special character)
GREY CAT	(contains a blank)

A program may use only those symbolic names which are utilized to label an element in the program. None of the labels may appear in the name field of more than one statement.

The value of a symbolic name specifying an instruction, constant, storage area, or any other element of the program is the address of the left-hand byte of storage field corresponding to this element. Therefore, the value of a symbolic name may be an integer non-negative number not greater than  $2^{24} - 1$ . Since relocating the program changes the addresses of its elements, the symbolic names are relocatable terms.

The *length attribute* of a symbolic name is a length (in bytes) of the storage field specified by this name. For instance, the label of an instruction, format RX, has a length attribute equal to four.

The *value to the location counter* may be used to facilitate references to the current instruction. During the assembly of any statement, the location counter contains the address of the left-hand byte of the program element described by this statement. To reference the value of the location counter use is made of the character \* (asterisk) which should be the first term in the expression. For example, the follo-

wing instructions place a zero code in the SIGMA field, if register 5 contains a binary number other than zero:

LTR	5,5
BC	8, * + 10
XC	SIGMA, SIGMA
. . . . .	

The BC instruction is 4 bytes long and the XC instruction, 6 bytes long, therefore, the expression \* + 10 defines the address of the instruction following the XC instruction.

Certainly, this instruction may be simply labelled by writing, say, the following instructions:

LTR	5,5
BC	8, LABEL
XC	SIGMA, SIGMA
LABEL . . . . .	

However, the use of an asterisk \* allows the program to be free from excessive labels.

The value of the location counter is a relocatable term. The length attribute of the counter is equal to the length of the instruction in which it is utilized. Therefore, in this example the length attribute of the term \* is equal to four.

The *self-defining term* is a term whose meaning is included in the term itself. In the previous sample examples we used self-defining terms, without obviously saying it, to specify the numbers of registers and mask of the BC instruction.

The Assembler language accepts four types of self-defining terms. These are decimal, hexadecimal, bit, and character self-defining terms.

The self-defining terms are used to specify register numbers, immediate operands in the instructions of the SI format, amount of shift in the shift instructions, fixed storage addresses, etc. Self-defining terms are absolute terms, i.e. they are not assigned a new value during a program relocating operation. The value of self-defining term must be an integer non-negative number not greater than  $2^{24} - 1$ . The other restrictions imposed on the values of self-defining terms are dependent upon their use. For example, a term indicating a floating-point register may take the values of 0, 2, 4, or 6, while a term specifying an amount of shift, the values from 0 to 63 (from 0 to 31 for single-precision instructions).

The *decimal self-defining term* is an integer decimal unsigned number. The decimal self-defining term is converted by the Assembler translator to its binary equivalent. Examples of decimal terms are as follows:

4095	0	255	004
------	---	-----	-----

The *hexadecimal self-defining term* is an integer hexadecimal unsigned number written in the form  $X'$  *hexadecimal-digits'*. Examples are  $X'80'$  or  $X'FFF'$ . The Assembler translator converts it to its binary equivalent. The maximum value of a hexadecimal term is  $X'FFFFFF'$ .

The *binary self-defining term* is a binary unsigned integer written in the form  $B'$  *binary digits'*, for instance,  $B'01101010'$ . A binary term may contain up to 24 bits.

The *character self-defining term* is a string of characters enclosed in quotes preceded by the letter  $C$ , for instance,  $C' + '$ . A character term may contain from one to three characters. It may include any characters of the EBCDIC (DKOI) code, except symbol' and ampersand characters. If a character term has to contain one of these characters ( ' and & ) they must be specified twice. For instance,  $C''''$  defines a single-character term', and  $C'A\&\&A'$  is a three-character term  $A\&A$ .

Character terms are commonly used to specify immediate operands in the instructions of the SI format. Each character in a character self-defining term is converted into an eight-bit equivalent.

One and the same value can be represented by different terms, and selection should be made of the term type most convenient for the purpose. For example, decimal term 13, hexadecimal term  $X'0D'$ , and binary term  $B'1101'$  have the same value. The decimal term is better to specify a number of a general register:

L     13,X

A mask in a BRANCH ON CONDITION instruction is better specified with the aid of a binary term. Thus, a statement

BC      $B'1101'$ , PROG2

looks more clearly than an equivalent statement

BC     13,PROG2

A hexadecimal term may be used to specify an immediate operand in the SI format, for example,

MVI     BYTE,  $X'0D'$

A *literal* is a constant specified in a machine instruction. When actions are performed on the operands in main storage, specified in the instruction are the addresses of the operands rather than the operands themselves. However, if one of the operands is a constant, the Assembler language allows us to code the constant itself in the instruction. During translation the constant is placed into a main storage field, and the address of this field is specified in the instruction.

The literal is coded as an ordinary constant (see Section 7, this Chapter) preceded by an equals sign = . For example, the instruction

IC     11, = X'FF'

places the hexadecimal constant FF into the low-order eight bits of general register 11.

The value of the literal is the address of a storage field in which the corresponding constant will be placed by the Assembler, for which reason the literal is a relocatable term. The length attribute of a literal equals the length (in bytes) of the storage field occupied by the constant.

Do not confuse literals with self-defining terms. The latter terms are intended for describing separate portions of an instruction. The use of literals simply cuts down the amount of work in coding a program, as the constants need not be described and labeled separately, and only then can be utilized in the statements.

The *reference to the length attribute* is a term whose value equals the length attribute of a symbolic name. To use the length attribute in expressions, we write the letter L and then a quote and the symbolic name the length of which is referenced to, for example L'GAMMA. The reference to the length attribute is an absolute term, for the length of any program element does not change during the program relocating operations.

As mentioned above, expressions are comprised by terms, arithmetic operation signs +, -, \*, / and parentheses according to the rules of Algebra. Expressions are used to represent operands in cases when it is impossible or inconvenient to represent an operand by one term. An expression was encountered above in the instruction

BC     8,\* + 10

(do not confuse the use of the character \* for a reference to the value of the location counter, and its use as a multiplication sign). Expressions are commonly used only for computation of addresses and have a fairly simple form, for example,

FIELD+256  
BETA-L'ALPHA  
A+(L'KA+L'KB) \* 10  
PL'Y+X'FFF'

The Assembler translator computes the value of the expressions by placing the values of terms in them and then performs the required actions following the rules of Algebra. The only exception is in that division by zero is allowed. The result of this division will be zero. Division always yields an integer, a fractional part, if any, is omitted.

Like the terms, the expressions may be absolute and relocatable, and have a length attribute which is taken equal to the length of the first (the only) term of the terms included in the expression. In this connection, the length attribute is conventionally defined also for those types of terms which have no actual length. The length attribute of self-defining terms and references to the length attribute is taken equal to one.

An expression is known as absolute, if its value does not change during the program relocating operations. An expression including solely absolute terms is obviously an absolute expression. However, there are absolute expressions which include relocatable terms too. An example is an expression

FIELD1 — FIELD2

in which both symbolic names specify different elements of a program. With the program at address 1000, let the address of field FIELD1 be 1150, and the address of field FIELD2 equal 1108, then the value of the expression is 42. If the program is moved as a whole, say to address 3000, then the field FIELD1 will be located starting at address 3150, and the field FIELD2, at address 3108. The value of the expression will remain the same 42.

Listed below are formal rules for expression computations.

1. An expression must not begin with a sign of arithmetic operation. For example, an expression  $-A + B$  is invalid and must be written in the form  $0 - A + B$  or  $B - A$ .

2. An expression should not contain more than 16 terms.

3. An expression should not include more than 5 levels of parentheses (a level is formed by a left-hand and its corresponding right-hand parentheses).

4. Do not use literals in expressions containing more than one term. Therefore, the instruction

IC     8, = X'FF'

is valid, and the instruction

IC     8, = X'FF' + 2

is invalid.

5. Only absolute terms and expressions may participate in multiplication and division.

6. The result of computing an expression should be a non-negative number, not greater than  $2^{24} - 1$ . Intermediate results may range from  $-2^{31}$  to  $2^{31} - 1$ .

The Assembler checks whether these rules are observed and issues messages on errors in the sample (program) listing.

### 7.4. Machine Instructions

In the Assembler language each machine instruction has its symbol which is similar to the instruction machine format, but does not duplicate it.

The Assembler compiler converts the symbolic representation of an instruction to the machine language placing the instructions at the halfword boundaries. Bytes remaining free in alignment to boundaries of halfwords are packed with zeros.

Any machine instruction may be labelled with a symbolic name. The length attribute of this name is dependent upon the instruction format: it equals two for the instructions in the RR format, four for the instructions in the RX, RS and SI formats, and the length attribute is equal to six for the instructions in the SS format.

The operands of machine instructions are registers, storage addresses, immediate operands, etc. In the symbolic statements depicting machine instructions, these operands are represented by absolute or relocatable expressions. The Assembler translator accomplishes certain checking of machine instruction operands. Thus, to represent a general register, use may be made only of an absolute expression assuming values from 0 to 15. An immediate operand must be specified by an absolute expression having a value from 0 to 255, etc. The Assembler translator also detects other errors such as use of a register having a number other than 0, 2, 4 and 6 in a floating-point instruction; a register with an odd number in multiplication, division, and double-position shift instructions; addresses of operands in main storage which are not aligned to the required boundaries, etc. Each error found causes printing of the corresponding message in the sample listing.

The rules for writing symbolic operands for the machine instructions of various formats are given below.

All the instructions of the RR format, except SPM and SVC instructions have the form

$$\text{OP} \quad R_1 R_2$$

The corresponding symbolic statements are written as follows:

$$[\text{label}] \quad \text{OP} \quad R_1, R_2$$

where  $R_1$  and  $R_2$  are absolute expressions having values from 0 to 15. Usually, to write operands in the instructions of the RR format use is made of decimal self-defining terms, for example,

$$\begin{array}{rcl} \text{SR} & & 7, 4 \\ \text{BLOCK 1 BALR} & & 14, 15 \end{array}$$

Here and hereupon the square brackets are used to denote optional elements of the symbolic statement format.

Since the SPM instruction does not use register  $R_2$ , it is in a symbolic format

[label] SPM  $R_1$

while the SVC instruction has the format

[label] SVC I

where I is an absolute expression having a value from 0 to 255 inclusive.

All the instructions of the RX format are represented in the machine code as follows:

OP  $R_1X_2B_2D_2$

while their corresponding symbolic statement is as follows

[label] OP  $R_1, D_2 (X_2, B_2)$

where  $R_1, X_2, B_2$  are absolute expressions (usually decimal self-defining terms) having values from 0 to 15, and  $D_2$  is an absolute expression having a value from 0 to 4095 (from  $X'0'$  to  $X'FFF'$ ). If  $X_2 = 0$ , no zero may be written, but the comma separating  $X_2$  from  $B_2$  must be used. Therefore, if the second operand is not indexed, the statement has the form

[label] OP  $R_1, D_2 (,B_2)$

If  $D_2 = 0$ , a zero must be written, for a statement of the form

[label] OP  $R_1, (X_2, B_2)$

is considered invalid. If the  $X_2$  and  $B_2$  are also equal to zero, we may omit all that is parenthesized

[label] OP  $R_1, D_2$

This method of writing operands which fully repeats the machine format of an instruction is called *explicit addressing*.

### Example

<i>Symbolic statement</i>	<i>Machine instruction</i>
L 13,4 (0,13)	58 D 0 D 004
CVB 6,0 (,1)	4F 6 0 1 000
BC B'1111', X'C'(14)	47 F 0 E 00C

The explicit addressing in the symbolic statements has but very little advantage over programming directly in the machine language, and is commonly used only in cases when a corresponding element of main storage cannot be labelled with a symbolic name, for instance, when this element is contained in another program and is transferred to a given program as a parameter.

Elements found in a given program are generally *implicitly* addressed by indicating the *symbolic name* the element is labelled with. In this event, the statement has the form

$$[label] \quad OP \quad R_1, S_2 (X_2)$$

and if  $X_2 = 0$ , the format of the statement becomes fairly simple:

$$[label] \quad OP \quad R_1, S_2$$

where  $S_2$  is a relocatable or absolute expression. The Assembler translator computes the value of the expression and splits it by itself into the base and displacement.

The instructions of the RS format have a machine format

$$OP \quad R_1 R_3 B_2 D_2$$

The corresponding statements of the Assembler language have the form

$$[label] \quad OP \quad R_1, R_3, D_2 (B_2)$$

or

$$[label] \quad OP \quad R_1, R_3, S_2$$

In the shift instructions, register  $R_3$  is not utilized, and the corresponding symbolic statements are written as follows:

$$[label] \quad OP \quad R_1, D_2 (B_2)$$

or

$$[label] \quad OP \quad R_1, S_2$$

Commonly, shifting order is specified by a decimal self-defining terms, for example,

STM	14, 12, 12 (13)	Save registers 14-12
SRL	5, 16	Shift logic register 5 by 16 positions right
BXH	5, 6, CYCLE	Branch on index high to label CYCLE

The instructions of the SI format having machine representation

$$OP \quad I_2 B_1 D_1$$

are written in the Assembler language in one of the following formats corresponding to explicit and implicit addressing of the first operand:

$$[label] \quad OP \quad D_1 (B_1), I_2$$

$$[label] \quad OP \quad S_1, I_2$$

Operands in the Assembler statements are written in their logical order, rather than in the order they appear in the machine instruction.  $I_2$  must be an absolute expression having a value from 0 to 255 (from  $X'O'$  to  $X'FF'$ ).

Examples of SI instructions:

NI	AREA1, X'F0'
CLI	BYTE, X'C4'
TM	SWITCH, B'01001001'
MVI	0(1), C' > '

In certain instructions of the SI format operand  $I_2$  is not used (LPSW, input/output instructions, etc.).

In the symbolic representation of these instructions operand  $I_2$  is also omitted:

[label]	OP	$D_1 (B_1)$
[label]	OP	$S_1$

The instructions of the SS format in which two length fields (lengths of two operands) given are represented in machine language as follows

OP     $L_1 L_2 B_1 D_1 B_2 D_2$

and the following formats of symbolic operands (for explicit and implicit addressing) correspond to this:

[label]	OP	$D_1 (L_1, B_1), D_2 (L_2, B_2)$
[label]	OP	$S_1 (L_1), D_2 (L_2, B_2)$
[label]	OP	$D_1 (L_1, B_1), S_2 (L_2)$
[label]	OP	$S_1 (L_1), S_2 (L_2)$

In these instructions, the operand record format may be selected independently for each operand.  $L_1$  and  $L_2$  are absolute expressions having values from 1 to 16 (usually decimal self-defining terms). In machine instructions the length attributes (length in bytes) take values from 0 to 15 and are always one less than the actual operand length. Specified in the Assembler operands is the actual length of operands, and in forming the machine instruction (code) the translator automatically subtracts one.

As with an address, the length of operands may be specified implicitly. The length attribute equal to the length attribute of an expression (term) defining the operand may be omitted. Operands with an implicit specification of length are written in the form

$D (, B)$  or  $S$

for explicit and implicit addressing, respectively.

Thus, each operand in the SS instructions can be written independent of one another following one of the

four methods:

D (L, B)      S (L)      D (, B)      S

and there are altogether 16 formats of symbolic statements representing these instructions.

The SS instructions with equal length of operands have the machine notation

OP   L   B<sub>1</sub>D<sub>1</sub>B<sub>2</sub>D<sub>2</sub>

while in the Assembler statements the length is written as part of the first operand:

[label]   OP      D<sub>1</sub> (L, B<sub>1</sub>), D<sub>2</sub> (B<sub>2</sub>)

L is an absolute expression having a value from 1 to 256. In the machine instructions (length code), the length attribute has a value from 0 to 255, and is always one less than the actual length of operands. Specified in the Assembler statements is the actual length of operands, and in forming a machine instruction the translator automatically subtracts one.

Use may be made of implicit addressing of each operand, and their length may be specified implicitly. In the Assembler language these instructions may be written in one of the following ways:

[label]	OP	D <sub>1</sub> (L, B <sub>1</sub> ), D <sub>2</sub> (B <sub>2</sub> )
[label]	OP	D <sub>1</sub> (, B <sub>1</sub> ), S <sub>2</sub>
[label]	OP	S <sub>1</sub> , S <sub>2</sub>
[label]	OP	S <sub>1</sub> (L), S <sub>2</sub> and so on

If the length of operands is omitted, then the Assembler translator utilizes the length attribute of the first operand.

### 7.5. Extended Mnemonics

In most programs fairly often use is made of the BC instruction. The use of the BC instruction requires the programmer to remember which mask is associated with which condition code and just what that condition code indicates in a given situation. To help the programmer in his work, the Assembler language includes extended operation mnemonics for the BC and BCR instructions. With extended mnemonics in use, only the second operand is written in the symbolic statement, while a mask of the `BRANCH ON CONDITION` instruction is formed by the translator against the operation code.

A list of extended mnemonics together with their machine instruction equivalents is given in Table 7.1. For the sake of clarity, the

Table 7.1

Extended code		Instruction with mask	Meaning
<i>General</i>			
B	S <sub>2</sub>	BC B'1111', S <sub>2</sub> '	Branch unconditionally
BR	R <sub>2</sub>	BCR B'1111', R <sub>2</sub>	Branch unconditionally (RR format)
NOP	S <sub>2</sub>	BC B'0000', S <sub>2</sub>	No operation
NOPR	R <sub>2</sub>	BCR B'0000', R <sub>2</sub>	No operation (RR format)
<i>After COMPARE instructions</i>			
BH	S <sub>2</sub>	BC B'0010', S <sub>2</sub>	Branch if operand 1 is high
BL	S <sub>2</sub>	BC B'0100', S <sub>2</sub>	Branch if operand 1 is low
BE	S <sub>2</sub>	BC B'1000', S <sub>2</sub>	Branch if operands are equal
BNH	S <sub>2</sub>	BC B'1101', S <sub>2</sub>	Branch if operand 1 is not high
BNL	S <sub>2</sub>	BC B'1011', S <sub>2</sub>	Branch if operand 1 is not low
BNE	S <sub>2</sub>	BC B'0111', S <sub>2</sub>	Branch if operands are not equal
<i>After arithmetic instructions</i>			
BO	S <sub>2</sub>	BC B'0001', S <sub>2</sub>	Branch on overflow
BP	S <sub>2</sub>	BC B'0010', S <sub>2</sub>	Branch if plus
BM	S <sub>2</sub>	BC B'0100', S <sub>2</sub>	Branch if minus
BZ	S <sub>2</sub>	BC B'1000', S <sub>2</sub>	Branch if zero
BNO	S <sub>2</sub>	BC B'1110', S <sub>2</sub>	Branch on not overflow
BNP	S <sub>2</sub>	BC B'1101', S <sub>2</sub>	Branch if not plus
BNM	S <sub>2</sub>	BC B'1011', S <sub>2</sub>	Branch if not minus
BNZ	S <sub>2</sub>	BC B'0111', S <sub>2</sub>	Branch if not zero
<i>After TEST UNDER MASK (TM) Instruction</i>			
BO	S <sub>2</sub>	BC B'0001', S <sub>2</sub> '	Branch if all ones
BM	S <sub>2</sub>	BC B'1100', S <sub>2</sub> '	Branch if mix
BZ	S <sub>2</sub>	BC B'1000', S <sub>2</sub>	Branch if all zeros
BNO	S <sub>2</sub>	BC B'1110', S <sub>2</sub>	Branch if not all ones
BNM	S <sub>2</sub>	BC B'1011', S <sub>2</sub>	Branch if not mix
BNZ	S <sub>2</sub>	BC B'0111', S <sub>2</sub>	Branch if not all zeros

address of the second operand is written implicitly, but, naturally, any other valid format may be used.

The examples that follow illustrate the use of extended mnemonics:

B	BEGIN	Unconditional branch to label BEGIN
BR	14	Unconditional branch to an address in register 14 (standard instruction of return from subprogram)
CLI	BYTE, C'A'	Branch to label ERROR if byte named

BNE	ERROR	BYTE contains no character A
TM	SWITCH B'011000 011'	Branch to label GOOD if all
BO	GOOD	tested bits of byte SWITCH contain ones
S	5, Y1	Branch to label POSITIVE if result
BP	POSITIVE	of subtraction operation is positive

## 7.6. USING and DROP Statements

As it has been mentioned in Section 4 of this chapter, implicit addressing of storage fields is splitted by the Assembler translator itself into the base and displacement. However, the translator cannot determine what registers are used in the program as base registers and what is their content. To this end, the Assembler language employs a special assembly control statement USING (*define the base register*) which is necessary in any symbolic program using implicit addresses. By means of this statement the programmer informs the translator (assembler) what general registers are intended for use as base registers for various areas of storage, and defines the values which must be in the registers at the time of the program execution.

The USING statement has the following format:

USING     V, R<sub>1</sub>, . . . , R<sub>n</sub>

where V is a relocatable or an absolute expression, and R<sub>1</sub>, . . . , R<sub>n</sub> are absolute expressions defining the numbers of the general registers. Obviously, *n* cannot be greater than 16, for the machine has only 16 general registers (in practice, more than one register are seldom specified in the USING statement).

The USING statement tells the translator (assembler) that general register R<sub>1</sub> contains value V, register R<sub>2</sub>, value V + 4096, register R<sub>3</sub>, value V + 8192, etc.

IMPORTANT! The USING statement does not load the base addresses into the registers. It only tells the assembler that the required values will be in the specified registers at the time of program execution. It is the duty of the programmer to allow for correct loading of the base registers. The USING statement produces no instructions or data in the object program. It serves solely to transfer information to the assembler.

**Example.** Commonly, a program in the Assembler language begins with the following statements:

```

BALR  9,0
USING *,9

BEGIN
```

In this example register 9 is selected as the base register (another register may be used). The BALR instruction loads the base value in register 9, and the USING statement tells the translator (assembler) that register 9 contains the address of the instruction following the BALR.

In this example the USING statement may be written in the form

USING BEGIN, 9

**Example.** During a reference to the subprogram, the address of the entry point must be in register 15. Therefore, it is enough to tell the translator (assembler) that the base address for the subprogram is in register 15:

```
USING    *, 15
STM      14, 12, 12 (13)
```

If the programmer wants to use another register in the subprogram as the base register, the beginning of the program may be as follows

```
USING    *, 11
STM      14, 12, 12 (13)
IR       11, 15
```

USING statements may be written anywhere in the program and as many times as necessary. The use of USING statements obeys the rules listed below.

1. Each implicit address in the program should be assigned a base register. A base register is known as available if the difference between the implicit address and the base value specified in the USING statement does not exceed 4095 bytes;

2. The USING statement applies to the implicit addresses in all symbolic statements following it, close to the end of the program, or to another USING statement defining a base register having the same number. In the latter case, the program must contain instructions modifying the value of the base as described in the new USING statement;

3. General register 0 may be specified in the USING statement as operand  $R_1$ , but in this event the value V is not used, and the translator (assembler) assumes that register 0 contains a zero value of base. If other base registers have been specified in the same USING statement, the assembler assumes that they contain values 4096, 8192, etc. A program containing such a USING statement will be not relocatable;

4. Register 0 is fetched as the base register for absolute implicit addresses less than 4096, though it may be not specified in the USING statements;

5. If several base registers are available for an implicit address, it will always take the one providing the least displacement.

The assembly control statement DROP (*Cancel the base register*) has the form

DROP      $R_1, R_2, \dots, R_n$

in which each operand defines one of the general registers. The DROP statement suppresses the use of each of the registers specified in it as the base register until it is encountered in the field of operands of the new USING statement.

### 7.7. Defining Constants

Before we can run a program on a computer, we need source data on which processing operations will be performed in addition to the machine instructions involved. Generally, the source data are input from external devices, but certain data may be inserted in the program itself. These data are known as *constants*.

The ES EVM set of instructions provides operations on data of diverse types: fixed- and floating-point binary numbers, packed and unpacked decimal numbers, symbolic and bit strings. Constants of diverse types may, respectively, be needed in programs. However, the machine can manipulate data of any type only in the hexadecimal form. Therefore, in machine language programming, much tedious work often has to be done in conversion of constants to machine internal representation. The Assembler language enables the programmer to write constants in a man-perceptible form, while their conversion to a machine language is accomplished by the Assembler translator.

To write constants in a program use is made of the assembly control statement *Define constant* (DC) which informs the translator (assembler) about the necessity to define a constant at the specified place of the object program. The DC statement has the following format:

[*label*]    DC   *constant(s)*

The label naming the constant receives the value of the address of the left-hand byte of the field occupied by the constant. The label characteristic length is equal to the field length occupied by the constant. Several constants can be defined at once in most of the DC statements. If that is the case, the value and label length attribute apply to the first of them.

The DC operand field consists of four subfields written close to one another with no delimiters:

[*duplication factor*] *type* [*modifiers*] *constant (s)*

The duplication factor causes the constant to be generated a specified number of times. Modifiers which are optional in a constant are described below separately for each type of constants.

The duplication factor and modifiers in writing a constant are optional, the type and constant itself are required.

The constant type is specified by one letter. For the types of constants, see Table 7.2.

Table 7.2

Type	Constant	Machine
C	Character	EBCDIC code; byte per character
X	Hexadecimal	One byte per each two hexadecimal digits
B	Binary (bit)	One bit per binary digit
P	Decimal (packed)	Decimal packed format; length from 1 to 16 bytes
Z	Decimal (unpacked)	Decimal zoned format; length from 1 to 16 bytes
F	Fixed-point	Binary number (usually a word)
H	Fixed-point	Binary number (usually a halfword)
E	Floating-point	Short floating-point number (usually a word)
D	Floating-point	Floating-point double-precision number (usually a double word)
A	Address	Address of main storage (usually a word)
Y	Address	Address of main storage (usually a halfword)
S	Explicit address	Address of main storage in the form 'base-displacement' (halfword)
V	External address	Address of an element from another program (usually a word)

The *character constant* is written in the DC statement in the form

*dCLn' characters'*

where *d* is the duplication factor, and *Ln* is the length modifier in which *d* and *n* are decimal self-defining terms. If no duplication factor is specified, the assembler (translator) forms one constant. Otherwise, *d* similar constants are formed successively in the storage fields. If the length modifier is omitted, the constant length in bytes is equal to the number of characters written. If the length modifier is specified, the constant then will be *n* bytes in length, and will be formed as follows:

- If the number of characters in the constant is less than *n*, the constant is blank-filled on its right;
- If the number of characters in the constant is greater than *n*, the rightmost characters are dropped.

The data are represented in the character constant using all characters of the EBCDIC code. As with the character self-defining terms, there is a particular rule for writing quotes and ampersand (&). Each of these characters must be written in the constant twice. One quote or ampersand will appear in main storage.

Regardless of the duplication factor, the maximum length of a character constant is 256 bytes.

### Example

<i>Assembler statement</i>	<i>Content of storage field in symbolic form</i>	<i>Label length attribute</i>
C1 DC C'JANUARY'	JANUARY	7
C2 DC CL9'JANUARY'	JANUARYbb	9
C3 DC 3CL2'JANUARY'	JAJAJA	2
C4 DC C'SMITH && SON	SMITH & SON	9
C5 DC 4CL2' * ' '	*b*b*b*b	2

Note that the duplication factor has no effect on the length attribute. Thus, in the above examples labels C3 and C5 have a length attribute of 2, though the fields formed in storage are of six and eight bytes, respectively.

The *hexadecimal constant* looks like the character constant, except that it is written in hexadecimal digits:

*dXLn'hexadecimal digits'*

Each two hexadecimal digits are translated into one byte. A leading hexadecimal zero must be added to a constant having an odd number of digits.

The actions of the length modifier, if specified, are as follows:

- If *n* is greater than the constant length, zeros are added on the left (leading zeros);
- If *n* is less than the constant length, the leftmost digits are dropped.

As the case is with the character constant, the maximum length of the hexadecimal constant is 256 bytes.

### Example

<i>Assembler statement</i>	<i>Content of storage field in hexadecimal form</i>	<i>Label length attribute</i>
X1 DC X'E1234'	0E 12 34	3
X2 DC XL4'FFF'	00 00 0F FF	4
X3 DC 3X'CA2'	0C A2 0C A2	0C A2 2
X4 DC 2XL3'47D'	00 04 7D 00	04 7D 3
X5 DC 2XL256'0'	00 . . . . . 00	(512 bytes) 256

The *binary constant* is written in the DC statement as follows

*dBLn'binary digits'*

The constant, type B, is in all aspects similar to the hexadecimal constant (type X) except that it is written in binary digits. Constants of the B type are good to specify various masks, bit switches, etc. If the number of binary digits (bits) written in a constant is not a multiple of eight, leading zeros are added to form a whole number of bytes. The length modifier, if specified, performs the same actions as in the constants, type X. The maximum length of a binary constant is 256 bytes.

The *decimal constants*, types P and Z are written in the DC statement as follows

*dPLn'number, number, . . . , number'*

or

*dZLn' number, number, . . . . , number'*

Several constants can be defined in one DC statement at the same time, the decimal numbers being separated from one another by commas. Each number is a sequence of signed or unsigned decimal digits. Constants, type P, are assembled into packed decimal fields, and type Z, into zoned decimal fields. The maximum length of any type constant is 16 bytes. If several constants are defined in the DC statement, the duplication factor and length modifier apply to each of them.

The length modifier, if specified, has the following effects:

- If *n* is greater than the constant length, leading zeros are added on the left in the packed or unpacked (zoned) format, respectively;
- If *n* is less than the constant length, the excessive digits are dropped.

### Example

<i>Assembler statement</i>	<i>Content of storage field in hexadecimal form</i>	<i>Label length attribute</i>
P1 DC P'1431'	01 43 1C	3
P2 DC PL3' +19, -345.7'	00 01 9C 03 45 7D	3
P3 DC 2PL2' - 1, +53.27'	00 1D 32 7C 00 1D 32 7C	2
Z1 DC Z'2745'	F2 F7 F4 C5	4
Z2 DC Z' -38, +9124'	F3 D8 F9 F1 F2 C4	2
Z3 DC ZL6'1'	F0 F0 F0 F0 F0 C1	6

In writing a decimal constant the programmer may mark the position of the decimal point within the constant. The written decimal point is ignored by the assembler. It does not affect the result.

The *fixed-point constant* (types F and H) is used to represent fixed-point binary numbers and is commonly written in the following form

*dFLn'number, number, . . . , number'*

*dHLn'number, number, . . . , number'*

Several fixed-point constants may be defined in one DC statement. If length modifier is not present, the implicit lengths are four bytes for the F type constants and two bytes for the H type constants. In this case a constant, type F, is aligned with the word boundary, and type H, with the halfword boundary. Positions of the bytes omitted during alignment are packed with zeros and are not treated as a part of the constant. With both types of constants the length modifier may indicate any length from one to eight bytes. If the length modifier is present, no boundary alignment is performed.

Each number in the constant is written as a signed or unsigned *decimal* number which may be followed (optionally) by a decimal exponent in the form  $Ew$ , where  $w$  is a signed or unsigned decimal integer. The number may include (optionally) a decimal point placed before, after, or within the number. Examples of writing fixed-point constants are as follows:

```

-123
  14.5E2
+27.5E +5
-1E -3
  743.E -2

```

During translation the numbers are converted to binary representation form and rounded to an integer.

For fixed-point constants *exponent* and *scale* modifiers may be specified in addition to the length modifier.

The exponent modifier written in the form  $Ew$ , where  $w$  is a signed or unsigned decimal integer number, causes the constant to be multiplied by  $10^w$  before conversion to binary representation. The exponent modifier performs the same functions as an exponent of a number, but applies to all constants in the statement, while the exponent refers to the number after which it is written.

The scale modifier written in the form  $Ss$ , where  $s$  is a signed or unsigned decimal integer, multiplies the constant by  $2^s$  after conversion to binary form (before rounding off). Actually, the scale modifier changes the position of the binary point in the number to allow use of fixed-point fractional binary numbers. However, defining such constants the programmer must himself see to it that the point is in proper position during execution of arithmetic operations.

Therefore, the general form of the constants, types F and H is as follows

```

dFLnSsEw' number, number, . . . , number'
dHLnSsEw' number, number, . . . , number'

```

All elements in defining a constant are written in the indicated sequence. In most cases no modifiers at all are required in writing fixed-point constants.

The *floating-point constants* (types E and D) are written in the DC statement in the following form:

*dELnSsEw' number, number, . . . , number'*  
*dDLnSsEw' number, number, . . . , number'*

The rules for writing numbers in fixed- and floating-point constants are similar.

If the length modifier is not present, the implicit length of a constant, type E, is supposed to be equal to four bytes, type D, to eight bytes. If that is the case, the constant, type E, is aligned with the word boundary, type D, with the double word boundary. With both types of constants, the length modifier may specify any length not in excess of eight bytes, the constant keeping a mantissa  $n - 1$  bytes in length. If the length modifier is specified, no boundary alignment is performed.

As with the fixed-point constants, the exponent modifier causes multiplication of all numbers specified in the statement by  $10^w$ .

Only positive scale modifier not in excess of 14 may be specified in the floating-point constants. Its action is as follows: after representation of floating-point data the mantissas of all numbers are shifted  $s$  hexadecimal positions to the right, and the exponents are corrected to save the valid value of the constants. Thus, the scale modifier permits defining non-normalized floating-point constants.

### 7.8. Address Constants and Defining Storage

The address constant is the address of a certain field of main storage which is used in the program as a constant. The address constants differ from all other types of constants in that the Assembler translator forms only conventional addresses of storage fields. The address constants are finally formed by the Editor when the program is arranged in an actual segment of main storage.

The *actual address constants* (types A and Y) are written in the program as follows

*dALn (expression, expression, . . . , expression)*  
*dYLn (expression, expression, . . . , expression)*

Several address constants may be defined in one DC statement. Unlike the other constants, the address constants are enclosed in parentheses, rather than in quotes.

The values of the expressions are calculated by the assembler and stored in the form of integer binary numbers. If the length modifier is not present, the implicit length of a constant, type A, is taken as equal to four bytes, and type Y, to two bytes. If that is the case, the constants, type A, are aligned to the word boundary, and type Y, to the halfword boundary.

If a relocatable expression is written in a constant, type A, then the explicit length can be specified equal to three or four bytes. Any length from one to four bytes may be specified for absolute expressions. If the length modifier is specified, no boundary alignment is performed.

For constants, type Y, an explicit length one or two bytes long may be specified, if an absolute expression has been written in the constant in question. If the expression is relocatable, an explicit length of two bytes may be specified. In this case, the length modifier is only to suppress the alignment to the halfword.

The constants, type Y, are similar to the constants, type A, in all aspects, but they find their applications commonly in small computers only, whose storage capacity is 32 768 bytes and less, for a halfword cannot accommodate a greater address.

Several written examples of actual address constants are as follows:

A1	DC	A(FIELD)
A2	DC	A(END -8, START +256)
A3	DC	AL1(52)
A4	DC	A(*+4)

The *explicit address constants* (type S) are used to define an address in the form 'base-displacement'. The constant, type S, always occupies a halfword. Alignment to the halfword boundary is performed or not performed, depending upon the presence of the length modifier which may be equal only to two bytes.

As with the address of a storage field in a machine instruction, the constant, type s, may be written in two ways: in the form of an absolute, or relocatable expression, or in the explicit form 'base-displacement'.

The assembler handles constants, type S, as it does with addresses in computer instructions. Here are some examples of written explicit address constants:

S1	DC	S (NAME)
S2	DC	S (BYTE + 4, X'FF' (13))

Generally the constants, type S, are used to define addresses of operands in machine instructions being formed.

The *external address constant* has the form

$dVLn$  (*external-name*, . . . , *external-name*)

The implicit length of the constant, type V, is supposed to be equal to four bytes. The alignment is made to the word boundary. The length modifier may specify a length of three or four bytes.

If the length modifier is specified, no word boundary alignment is performed.

The *external name* is a symbolic name defined in another program.

The assembler cannot specify the value of this name, for which reason the value of the constant after assembly will be equal to zero. The final value of the external name is set during the stage of editing, for which purpose the Assembler translator forms a directory of external symbols. The directory is later used by the Linkage Editor.

The constants, type V, are mainly used to specify the address of entry to a subprogram assembled separately.

**Defining storage.** Any program should have working areas to store data. The initial value of these areas is of no importance, the assembler must be only told their arrangement and length. There is a special means in the Assembler language for reserving storage areas and assigning them symbolic names known as the DS (*Define Storage*) statement. The format of the DS statement is similar to the format of the DC statement, but according to the DS statement, the assembler (translator) forms nothing in storage. It simply reserves the required number of bytes in the specified place of the program. Therefore, there are certain distinctions between the rules for writing operands in these statements.

- No scale and exponent modifiers are used in the DS statement;
- The maximum length specified by an operand, type C or X, is equal to 65 536 bytes (against 256 bytes in the DC statement);
- The constant in the DS statement may not be written. In this case, the size of the storage field reserved is defined by the length modifier;
- If both the constant and length modifier are omitted, the implicit length is equal to one byte for operands, types C, X, B, P, and Z, two bytes for operands of type H, four bytes for operands of types E and F, and eight bytes for operands, type D.

The boundary alignment for the storage areas defined by the DS statement is performed as dictated by the operand type, like the case is with the DC statement. The distinction is in that the bytes omitted in the alignment are not filled and their contents during the program execution are unpredictable. Like in the DC statement, no boundary alignment is performed, if the length modifier has been specified.

Some sample examples of using the DS statement are as follows:

L1	DS	C	Reserve one byte
L2	DS	2H	Reserve two halfwords, perform alignment to halfword boundary
L3	DS	XL1024	Reserve a field 1024 bytes long. Type C may also be specified in this statement
L4	DS	F	Reserve word, perform alignment to word boundary
L5	DS	FL4	Reserve four bytes, do not perform alignment
L6	DS	CL80	Reserve area 80 bytes long
L7	DS	80C	Reserve 80 areas, each one byte long

The statements with labels L6 and L7 reserve one and the same field of storage. However, the label L6 has a length attribute equal to 80, and label L7, to 1, because the duplication factor does not affect the length attribute. This should be taken into account in the SS instructions where use is made of implicit length operands.

**Zero as duplication factor.** A zero as the duplication factor of an operand may be specified in the DS statement. According to the statements with a zero as the duplication factor, no storage areas are reserved. However, the labels of such operands receive the same length attribute and value as those used for other than the zero duplication factor.

If no length modifier is specified in a statement with a zero as the duplication factor, alignment is performed to the boundary defined by the operand type. In this case, omitted bytes may contain any information.

If the length modifier is specified in a statement with a zero as the duplication factor, no boundary alignment is made, and this statement is intended only for assigning a value and length attribute to a symbolic name.

Presence or absence of other modifiers and the constant itself in the statements with a zero as the duplication factor does not affect the result.

Basic applications of statements with a zero as the duplication factor are illustrated by the two examples given below.

**Example.** Reserve a field, 800 bytes long, to store 200 fixed-point numbers. To use a statement of the form

```
ROW    DS    CL800
```

for the purpose would be incorrect, because the field will not be aligned with the word boundary. One of the correct solutions is

```
        DS    0F
ROW    DS    CL800
```

Alignment of storage areas to the required boundaries is performed in the Assembler language most often with the aid of statements having a zero duplication factor.

**Example.** Suppose that a program uses the current date stored in a six-byte field in the form  $\overline{DDMMYY}$ , where  $\overline{DD}$  (DD)—day, MM—month, and  $\overline{YY}$  (YY)—two last digits of the year. If actions are performed in the program on the whole of the date and on its separate fields, the area for date storage is best described as follows:

```
DATE    DS    0CL6
DAY      DS    CL2
MONTH    DS    CL2
YEAR     DS    CL2
```

The first DS statement only names the whole area, since a zero duplication factor is specified in it. The DATE label has a value equal to the address of the first byte of the storage area, and a length attribute of six. The other DS statements describe separate two-byte fields of the area. With this description, use may be made in the SS instructions only of labels without explicitly specifying the length, for each of these labels has a required value and a needed length attribute.

**CNOP statement.** Boundary alignment may be required not only in data areas or work fields, but also in area of instructions. Examples are subprogram calls written in the Assembler language.

Recall that in calling a subroutine, the address of the subroutine entry point must be loaded in register 15, the address of return point in register 14, and the address of the table containing parameter addresses in register 1. The address of the subprogram save area is generally loaded in register 13 at the beginning of the program. Programming experience shows that the parameter address table is better stored together with the subroutine call instructions rather than in a separate area of storage. This makes the program clearer.

Let, for instance, the subroutine entry point have a symbolic name SUBROUT, and the parameters transferred to the subroutine be furnished with labels X, Y, Z. Consider the following sequence of Assembler statements:

L	15,=V (SUBROUT)
LA	14,RETURN
BALR	1, 15
DC	A(X, Y, Z)

RETURN

The first instruction loads the address of the subroutine entry point in register 15. This instruction uses an external address constant written in the form of a literal.

The LA instruction loads the address of the return point from the subroutine (label RETURN) in register 14. The BALR instruction passes control at the address in register 15, storing at the same time the address of the storage byte following this instruction in register 1. The last symbolic statement defines three address constants corresponding to the subroutine call parameters.

At first superficial sight of the above instructions, it may seem to one that the execution of these instructions will result in a correct subroutine call. However, there is an error in this segment of the program which is not so easy to perceive. The matter is that the machine instructions are aligned by the assembler on the halfword boundary, and address constants, type A, on the word boundary. The-

refoe, skipped bytes may appear between the BALR instruction and the constants, e.g. the BALR instruction is two bytes long and, if it is aligned on the word boundary, two zero bytes will be skipped till the next word boundary the address constants will be aligned on. Should it be that the BALR instruction is aligned on the second halfword boundary in the word, then the address constants will follow this instruction directly and the program will run properly. Therefore, the above-given segment of the program must be provided with certain statements to make the BALR instruction be always aligned on the second halfword boundary in the word. To obtain this by the above means of the Assembler language is impossible, since in the boundary alignment of the DC statements, the skipped bytes

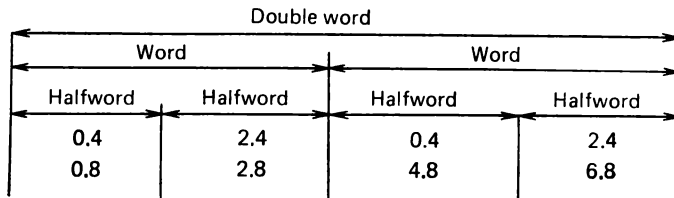


Fig. 7.1. CNOP statement aligning on boundary

are zero packed. A zero field in the flow of instructions will cause a program interrupt on an invalid op-code, for the machine has no instruction with a zero op-code.

For boundary alignment in the instruction area, the Assembler language is provided with a **CONDITIONAL NO OPERATION (CNOP)** instruction which has the format

CNOP      *a, b*

where  $a, b = 0, 4; 2, 4; 0, 8; 2, 8; 4, 8; 6, 8$ .

The  $a$  operand shows on what byte of the word (for  $b = 4$ ), or double word (for  $b = 8$ ) the program element following the CNOP statement must be aligned. For example,  $a, b = 0, 4$  defines word boundary alignment, and  $a, b = 4, 8$  defines alignment on the second word boundary in a double word, etc. The positions within a double word which are defined by each pair  $a, b$  are shown in Fig. 7.1.

If the location (address) counter is set to the desired boundary, the CNOP statement changes nothing, otherwise the skipped bytes are filled with NOPR (NO OPERATION) instructions, i.e. hexadecimal code 0700 is loaded in each halfword skipped. If the location counter is set to the boundary of an odd byte, then before processing the CNOP statement, the assembler (translator) itself skips one byte packing it with zero code.

With the aid of the CNOP statement, a subroutine call is written as follows:

CNOP	2,4
L	15,=V (SUBROUT)
LA	14,*+6+4*3
BALR	1, 15
DC	A (X, Y, Z)

Now the reader sees that the last sample example is free from errors. In order not to overcrowd the program with many labels, the second operand of the LA instruction is written in the form of an expression containing the values of the location counter.

### 7.9. Program Sectioning

Because of complex problems being solved by Assembler language programs they are generally long and cumbersome. Seldom such a program is executed and written by one person. Programming practices have shown that it is more convenient to divide large programs into separate, relatively independent program sections each of which solves some portion of the problem posed.

The Assembler language allows program sections to be written and assembled (translated) either together, or separately from one another. Separate program sections are finally interlinked into an executable single program by the Linkage Editor. In order to allow the Linkage Editor to find correct links between the sections, an *external symbol directory* is included by the Assembler translator into each object module.

The program sectioning is optional and the sectioning facilities dealt with in this section are commonly utilized only in large programs. Linkage means, however, are often used in simple programs, because in building almost any program, use is made of standard subroutines which actually are program sections written by other persons and separately translated.

The input to the Assembler translator is the source module which may include one or several program sections. Sections of different types in the source module are defined and separated from one another with the aid of the following Assembler statements:

START	— start source module
CSECT	— identify control section
DSECT	— identify dummy section
COM	— identify common control section
END	— terminate source module

The START statement is used to name the first (the only) control (program) section of the source module, and to set the initial value

of the location counter. The START statement has the following format:

[label] START [self-defining term]

The label naming the START statement is accepted as the name of the control section. If no label is present, the START statement defines an *unnamed control section*. The Assembler translator uses the self-defining term in the START statement as the initial value of the location counter. This value should point to the boundary of a double word, i.e. be a multiple of eight. The initial value of the location counter is taken equal to zero, if this operand is omitted. The address defined by the START statement is conventional, for the Linkage Editor can, if necessary, relocate any program in another storage area.

The START statement is optional, and if it is not present, the assembler considers that written in the program is the statement

START 0

The START statement must be the first statement in the program. All statements that follow are translated as part of the control section defined by them, until one of the statements CSECT, DSECT, or COM defining another section is encountered.

The CSECT statement has the following format:

[label] CSECT

The operand field in the CSECT statement is not used. If no symbol is written in the label field, the section is considered unnamed, otherwise the label names the control section. All statements following the CSECT statement are translated as part of the section defined by it, until another CSECT, DSECT, or COM statement defining another section is encountered.

The Assembler translator gets its own location counter for each program section, starting from the boundary of the double word following the last used address of the previous section.

The Assembler language provides convenient facilities for use of storage areas reserved in other programs. To this end, use is made of the DSECT statement which has the following format:

[label] DSECT

The DSECT statement should always have a name. No operand fields are present in it. The DSECT statement is handled by the assembler exactly in the same way as the CSECT statement, except that no storage space is allocated for the elements found in the dummy section. It is supposed that storage for them is reserved in another source module, or in another part of the source module.

Let a storage area in the form of a control section be defined in the program:

FIELDX	CSECT	
X1	DS	CL15
X2	DS	CL4
X3	DS	3F
X4	DS	CL80

In order to execute the operations with fields X1 through X4 in another program, one may load constant V (FIELDX) in a general register, say register 9, calculate the displacement of each field relative to the beginning of the section, and use explicit addresses of the form 15 (9) for field X2, or 31 (9) for field X4. The program will run correctly. However, it will become less clear which is characteristic of a symbolic language. More than that, if for some reason the structure of the FIELDX area has to be modified, then it will be necessary to recalculate all the displacements and correct all instructions in the other program which are involved in manipulating the fields of this area.

It is much better to use a dummy section for the purpose. Having written in the other program statements of the type

```

. . . . .
                USING    FIELDY,9
                L        9,=V (FIELDX)
. . . . .
FIELDY  DSECT
Y1      DS      CL15
Y2      DS      CL4
Y3      DS      3F
Y4      DS      CL80
. . . . .

```

and using labels Y1 through Y4 in the machine instructions, we shall in fact reference fields X1 through X4 defined in the first program.

In each source module, the COM statement can define one unnamed common section. The common sections defined in each of the source modules constituting the program will be assigned, as a result of editing, the same space in storage. If the elements of the common section are described in a similar way in all source modules, then in references to the common section from any module, operations will be executed on the same data.

The last statement in any assembly program must be the END statement which stops the translation of the source module. Its format is as follows

```
END    [relocatable-expression]
```

An operand, if specified, defines the instruction with which the execution of the program will start. Generally, an operand is specified in the END statement, if the execution of the program starts with other than the first instruction. The END statement must be used in each source module.

The *program linkage means* allow symbolic names defined in other modules to be used in the source modules. We know already one of the linkage means. This is the V-type address constant specifying the address of an external name. External names can also be defined in an EXTRN sentence which has the following format:

EXTRN     *external-name<sub>1</sub>, external-name<sub>2</sub>, . . . , external-name<sub>n</sub>*

The names present in constants, type V, should not appear in sentences EXTRN. Names from the EXTRN sentence (statement) may be used as labels in the program; however, each of them must be assigned, by the statement USING, its base register with the required value in it. Owing to this, the EXTRN statement is used fairly seldom, external names being commonly defined in constants, type V.

In translation of a source module the translator (assembler) does not know what labels of those defined in the module are used in other modules as external names. To include all labels appearing in the program in the External Symbol Directory is not advantageous, since most of them are not used in other programs. So, in the Assembler language the following convention is accepted: only names of control sections are automatically included in the External Symbol Directory. Additional names used in other source modules must explicitly appear in the ENTRY statement which has the following format:

ENTRY     *name<sub>1</sub>, name<sub>2</sub>, . . . , name<sub>n</sub>*

The operands of the ENTRY statement must be symbolic names defined in the named control sections. No labels from dummy, common, or unnamed control section may appear in the ENTRY sentence.

## 7.10. Macros

The Assembler language features considered above allow a program to be written in the form of a sequence of symbolic statements depicting machine instructions, and Assembler statements used to transfer certain information to the translator. Each machine instruction is shown by a symbolic statement, and since the machine instructions perform only simple operations (addition or multiplication of two numbers, transfer of control, etc.), the programs in the Assembler language are rather long.

One of the methods of shortening the length of symbolic programs is known to us. This consists in setting aside standard parts of the problem and use of standard subroutines. In various programs, however, repeated sequences of statements are often encountered which cannot be reasonably grouped into subroutines. An example is represented by the subroutine call instructions described in Sect. 7.8:

CNOP	2,4
L	15,=V (SUBROUT)
LA	14, *+6+3*4
BALR	1,15
DC	A(X, Y, Z)

This typical subroutine call consists of five symbolic statements. To call any other subroutine, it is necessary to write the same statements CNOP, L, LA, BALR, and DC in which only boldly printed parts above are modified. It is clear that the method of cutting down the program length will not do here. Indeed, what an absurd attempt would be to write a subroutine call in the form of a subroutine (for you will have to write a call for it too).

The solution found by the designers of the Assembler language is best explained by the following example. In many scientific books certain concepts and expressions are frequently repeated. In order not to repeat frequently the same words in a text, a convention is used to write such words in the form of acronyms, abbreviations, or conventional signs, and supplement the book with a list of conventional signs explaining the meaning of each of them. The macros included in the Assembler language allow the programmers exactly in the same way to introduce their own abbreviations for frequently repeated sequences of statements.

Given below is a small glossary of various terms pertaining to the macros of the Assembler language.

*Macro instruction*—A condensed form for certain sequence of symbolic statements.

*Macro definition*—The set of statements defining the name, format and generating conditions for a given *macro instruction*. Each macro instruction in use must have its own macro definition which tells the Assembler translator what symbolic statements are to be inserted in the program in place of the macro instruction.

*Macro language*—Representation and rules for writing *macro instructions* and *macro definitions*.

*Macro library*—A set of various macro definitions used in programs recorded on a magnetic disk.

*Macro generation*—The process of substituting conventional symbolic statements for a macro instruction in compliance with the macro definition.

*Macro expansion*—A set of symbolic statements appearing in the program in place of a macro instruction as a result of the macro generation.

The translation of the symbolic program is carried out in two steps. The first step is macro generation. A corresponding macro definition is found for each macro instruction included in the program. As the next step, the Assembler translator, following the rules of the macro language, builds up a macro expansion which replaces the source macro instruction in the program text. Upon completion of the macro generation, none of the macro instructions is left in the program which contains only machine instructions and Assembler statements. The second step is the assembly (translation) itself.

As a rule, macro definitions are written by skilled programmers. At present, the Assembler Macro Library includes a great number of various macro definitions. Unfortunately, lack of space in the book does not permit us either to describe the macro language, or to consider in detail even most usable macro instructions for which reason we shall limit ourselves to superficially considering several macro instructions of the Disk Operating System which will be used in the next section of the book. For more detailed information about the macros of the Assembler language, see the description of the ES EVM Operating System.

**CALL.** This macro instruction is used to initiate a call to a subroutine. It has the following format:

[*label*] CALL *entry-point-name* [(*list of parameters*)]

Where the *entry-point-name* is an external name defining an entry point to the subroutine, and the *parameter* is an expression specifying the address which must be transferred to the subroutine.

For example, as a result of the macro instruction

CALL SUBROUT (X, Y, Z)

appearing in the program, the above-mentioned macro expansion will be generated. Certainly, this macro expansion may be written by the programmer himself, but the use of the macro instructions saves space and makes the program clearer.

**SAVE.** This macro instruction is used to save the contents of the registers of the external program. The macro instruction format is as follows:

[*label*] SAVE ( $R_1$  [,  $R_2$ ])

where  $R_1$  and  $R_2$  are self-defining terms, specifying the numbers of general registers. If no register  $R_2$  is specified, the content only of one register  $R_1$  is stored. If both numbers are specified, the contents of all registers found between  $R_1$  and  $R_2$  are saved. Thus, the macro

instruction

BEG SAVE (14, 12)

produces a macro expansion consisting of one statement only

BEG STM 14, 12, 12 (13)

This macro instruction cuts down the record very little, but it saves programmer's brainwork who has not to remember in what word of the save area the content of each register is stored.

**RETURN.** Restoration of the contents of registers and exit from the subroutine can be accomplished with the aid of this macro instruction having the format:

[label] RETURN (R<sub>1</sub> [, R<sub>2</sub>])

The macro expansion produced by the RETURN macro instruction is fairly simple, too. For instance, the macro instruction

RETURN (14, 12)

is substituted for in the program by the following statements:

LM 14, 12, 12(13)

BR 14

The SAVE and RETURN macro instructions require that register 13 contains the address of the save area.

The three above-mentioned macro instructions are known as *linkage macro instructions* for linkage between programs. In this section we shall touch on one more class of macro instructions—*input/output macro instructions*.

As it has been mentioned in Chapter 5, the ES EVM system must include an Input/Output Control System (IOCS) which is a set of program which organize the whole of the information interchange work between peripheral units and the central processor.

The IOCS comprises two parts called in another words levels of access to external information: a *physical* level and a *logical* level. At the physical level of access the programmer should write channel programs himself. The physical IOCS organizes interchange operations with the I/O devices according to these channel programs without going carefully into the structure and contents of the information being transferred. The IOCS physical level is generally used when the required interchange work cannot be carried out with the help of the logical level programs, or when the logical level programs carry out it too slowly.

While at the physical level the programmer manipulates the channel commands, status bytes and sense bytes, etc., the programs

of the logical IOCS relieve him of these actions. At the IOCS logical level the principal terms are files, blocks and records.

A *file* is an organized collection of data on an external medium, for example, a card deck, a printed document, several sequential zones on magnetic tape, etc. In the programming system adopted in the ES EVM, each file must be provided with special labels containing information on the purpose of the data contained in the file. The label structure of files varies with the type of the I/O device. With storage devices such as magnetic tape and disk units, fairly complicated labels are required, since one spool of magnetic tape or disk pack can incorporate many files, and the logical IOCS must be capable of selecting a required file of them. With punched card files use is made of only one very simple end-of-file label which is an end-of-file card containing characters /\* in the first two columns. The logical IOCS recognizes the card as indicating end of card deck. No labels at all are used with files on printers, though, it is not good practice to print any information without heading and destination data, because with many users, it often takes the computer operator much time to detect the owner of unnamed output.

A *block* in a group of characters, words, or records placed in an input/output medium, examples being a print line or magnetic tape zone. A block may contain one or more logical records.

A *logical record* is a record containing all the items necessary to represent some specific functions. For example, in a magnetic tape file containing information about the workers of an enterprise, the data pertaining to a worker form a logical record.

If each file block contains one logical record, the records are known as *unblocked*, otherwise, they are called *blocked* records.

The point of blocking logical records is in accelerating the program work, because several logical records are read from, or written onto, in one call to an I/O device. Besides, the efficiency in storage space increases, since the number of block-to-block gaps on magnetic tape or disk decreases. A decrease in the program work time must be as usual paid for by storage space, as the amount of input/output areas in main storage which the information is interchanged with has to be increased.

With the physical IOCS, grouping of logical records into blocks at the output, or selecting of separate records from a block during input should be accomplished by the programmer himself. At the logical level this is one of the IOCS functions.

To make a call to the IOCS programs easier, included in the Assembler macro library are input/output macro instructions which are divided into declarative and imperative.

*Declarative (descriptive)* I/O macro instructions (macros) are used to define files. These instructions are used to generate *file description tables* containing, for instance, such information as the type of I/O

device which carries the file, the address of the area to locate a successive record, length in bytes of the block and logical record, etc. All files that are to be processed must be described.

*Imperative* I/O macro instructions generate calls to the programs of the logical IOCS performing actual input/output operations.

This section deals with the declarative macros for card and print files, and also with the imperative macros for the input/output of recurrent logical record.

**DTFCD.** This macro defines the file for card. Its format is as follows:

*File-name DTFCD operands*

The *file-name* is a conventional symbolic name with the only restriction that it must contain from one to seven (rather than eight) characters. *Operands* in the declarative macros have the so-called keyword format:

*keyword = parameter*

Operands may be written in any sequence with commas between them. Certain operands must be used in all files, while others are used only when the programmer needs somewhat additional processing of the file.

Given below is a list of most frequently used operands of the DTFCD macro instruction.

Operand	Functions
DEVADDR = SYSnnn	<i>Device Address.</i> This is the address of the I/O device carrying the file.
BLKSIZE = n	<i>Blocksize.</i> It indicates the size (in bytes) of the input/output area for accommodation of the file blocks. If this parameter is omitted, 80, i.e. a complete card, is assigned.
IOAREA1 = name	<i>Input/Output Area.</i> This is the name of the input/output area.
TYPEFLE = INPUT OUTPUT	<i>Type of File.</i> This indicates the type of the file (input or output). If omitted, INPUT is supposed.
WORKA = YES	<i>Work Area.</i> This indicates a separate work area to which a recurrent record is transferred. If the program processes records directly in the input/output area, the operand is dropped.
EOFADDR = name	<i>End of File Address.</i> This is the label to which control must be passed on end-of-file. It is required only for input files.

The meaning of each operand of the DTFCD macro instruction is fairly clear, and we shall dwell only on the DEVADDR operand. This operand indicates the *logical address* of the I/O device. It is not convenient to specify in the program physical addresses of I/O

devices in the form channel-device, for a machine may have several I/O devices of the same type, and it may happen that a required device will be engaged or defective, meanwhile the machine may include another similar device free and in good condition, but with a different address. Besides, similar numbers may be assigned to different devices on diverse machines, and, say, a card reader of one machine may have the same number as a printer of another machine.

Because of this, the ES EVM programming system permits specifying only conventional or logical addresses of devices, and the conformity between the logical and physical addresses is established only before execution of the program.

The logical addresses are written in the form `SYSnnn`, where `nnn` is a three-digit number lying within 000 to a certain maximum number permitted by the machine supervisor (commonly it is 244). The logical devices from `SYS000` to `SYS244` are called the logical devices of the programmer. Besides, the programmer may utilize system logical devices some of which are described below:

- `SYSIPT` — a system input device (commonly a card reader).
- `SYSPCH` — a system card punch (commonly a card punch).
- `SYSLST` — a system output device (commonly a line printer).
- `SYSLOG` — a system device of communication with the operator (commonly a console typewriter).
- `SYSRES` — system residence which is a device on which the operating system resides (only disk).

The system logical devices may not conform to the above-mentioned physical devices. For example, magnetic tape may be well assigned to `SYSLST` or `SYSPCH` to obtain the results of program execution later on paper or punched cards. The programmer should not worry about this, because whatever the assignment may be, the program is executed in a similar way.

An example of describing an input file on punched cards is as follows:

```
INCARD DTFCD  DEVADDR = SYSIPT, BLKSIZE = 80.           X
                IOAREA1  = FIELD1, TYPEFLE = INPUT.      X
                EOFADDR = ENDCARD
```

The X character in the rightmost column of the record on the form must be in the continuation column. The same macro instruction may be written in another way, omitting optional operands:

```
INCARD DTFCD  IOAREA1  = FIELD1,      X
                EOFADDR = ENDCARD,    X
                DEVADDR = SYSIPT
```

These macro instructions mean that the card file will be read from the system logical device `SYSIPT` into the storage area named

FIELD1, and at the end of the file control will be passed to the label ENDCARD. The field FIELD1 must be described in the program:

```
FIELD1 DS CL80
```

**DTFPR**—*Define the print file.* It has the same format as DTFCD:

```
file-name DTFPR operands
```

The most usable operands of the DTFPR macro instruction are as follows:

Operand	Functions
DEVADDR = SYSnnn	Required for each file
BLKSIZE = n	Required, if the print line length is not equal to 121 characters
IOAREA1 = name	Required for each file
WORKA = YES	Required in use of a separate work area in which a print line is formed. If a line is formed in the input/output area, the operand is dropped

The meaning of each operand is the same as in the previous macro instruction. An example of the DTFPR macro instruction is as follows:

```
OUTPRIN DTFPR DEVADDR = SYS005, BLKSIZE = 128,    X
          IOAREA1 = FPR, WORKA = YES
```

The imperative I/O macro instructions are divided into macros of preparation and completion, and macros of processing.

A file must be opened (prepared) before it is processed. This can be done with the help of the imperative macro OPEN having the following format:

```
[label] OPEN file-name1, file-name2, . . . , file-namen
```

One OPEN macro instruction can open from 1 to 16 files. The actions performed on an OPEN macro are dependent upon the type of the file. Labels are checked at this time for magnetic tape input files, file labels are created for output files, and only a 'file open' flag is set for print files in the file description table, etc.

Upon completion of processing any file must be closed which is accomplished by the imperative macro CLOSE. Its format is as follows:

```
[label] CLOSE file-name1, file-name2, . . . , file-namen
```

The macro creates ending labels for output files and clears the 'file open' flag in the file description table for print and input files.

File processing is performed with the aid of the GET and PUT macro instructions which have the following format:

```
[label] GET file-name [, work-area-name]
```

```
[label] PUT file-name [, work-area-name]
```

The GET macro instruction places a recurrent record of the file being input in the input/output or work area. The name of the work area must be specified in the GET macro only when a WORKA = YES operand has been specified in the DTF macro for this file.

The recurrent record of the output file is output to an external storage unit by the PUT macro instruction. Likewise, the name of work area in the PUT macro may be specified only for files having a WORKA = YES operand.

Examples of the imperative macros for the above-described files are as follows:

1. Open both files:

OPEN INCARD, OUTPRIN

2. Place a recurrent record of the INCARD file in the FIELD1 area:

GET INCARD

3. Print a record of the OUTPRIN file performed in the LINE2 field:

PUT OUTPRIN, LINE2

4. Close both files:

CLOSE INCARD, OUTPRIN

In their course of work the IOCS programs utilize general registers 0, 1, 14, 15. Besides, register 13 commonly contains the address of the save area. Therefore: (a) the contents of registers 0, 1, 14, and 15 must be saved in the subprograms containing the macro instructions, and (b) the programmer may use without any risk only registers 2 through 12.

### 7.11. Sample Programs

This section considers several sample programs in the Assembler language. The examples are of study nature, i.e. they do not solve any actual problems, but are used to illustrate certain techniques of programming in the Assembler language.

The Reader is recommended well go into how each of the programs that follow will be executed.

**Example.** Let us consider a program for printing a table of square roots of integer numbers from 1 to 100 accurate to three digit positions after the decimal point. To calculate  $X = \sqrt{N}$  use is made of Newton's method. First, let  $X_0 = N$ , and then calculate  $X_1, X_2, \dots$  by the formula

$$X_{k+1} = 0.5(X_k + N/X_k)$$

The calculations are continued until the difference between two sequential values of X is less than 0.0001. Next, the obtained value of X is rounded off to three places after the decimal point, and the values of N and X are printed.

```

1.          START
2.  PRINT    DTFPR  DEVADDR = SYSLST,          X
3.          BLKSIZE = 16, IOAREA1 = 'Y, X
          WORKA   = YES
4.  B1       BALR   12,0
5.          USING  *,12
6.          OPEN   PRINT
7.          PUT     PRINT, HEAD
8.          ZAP     N,=P'1'
9.  B2       ZAP     X, N
10.         MP      X,=P'10000'
11.         ZAP     ND, X
12.         MP      ND, =P'10000'
13.  B3       MVC    XD, ND
14.         DP      XD, X
15.         AP      XD(5), X
16.         MP      XD(5), =P'5"
17.         MVO     XD(5), XD(4)
18.         CP      XD(5), X
19.         BE      B4
20.         ZAP     X, XD(5)
21.         B       B3
22.  B4       AP      X, =P'5'
23.         MVC     FLD, PICTURE
24.         ED      FLD, N
25.         PUT     PRINT, FLD
26.         AP      N, =P'1'
27.         CP      N, =P'100'
28.         BNH     B2
29.  B5       CLOSE  PRINT
30.         SVC     14
31.  HEAD     DC      CL16'bbbbNbbbbSQRT(N)'
32.  PICTURE  DC      X'40404020202022202020204B202020'
33.  Y        DS      CL16
34.  FLD      DS      CL16
35.  N        DS      PL2'100'
36.  X        DS      PL5'100.0000'
37.  ND       DS      PL10'100.00000000'
38.  XD       DS      PL10
39.          END     B1

```

The numbers in the left-hand column do not apply to the program. They are only for reference purposes. To illustrate techniques of calculations on fractional numbers we use decimal arithmetic instructions in the program.

Decimal integer numbers are stored in the N field. The X field contains numbers with four digits after the suspected decimal point, and the ND field — with eight digits after it. Therefore, after moving N into X and X into ND the numbers must be multiplied by  $10^4$  in order to preserve the position of the suspected decimal point. Constants are written in statements Nos. 36, 37 and 38. These are optional and serve to show the programmer the position of the suspected decimal point.

After decimal division in statement 14 the quotient will occupy the five left-hand bytes of the XD field, for which reason, the length should be explicitly specified in the subsequent statements.

Statements 16 and 17 divide the sum  $X + N/X$  by 2 by multiplying it by 5 and dividing by 10. Dividing by 10 is carried out by means of the MVO instruction.

Statement 22 rounds off the result to three places after the point, adding 5 to the next position.

Before printing the result is edited. The PICTURE field contains a pattern *bbbddd f d d d d d*. *ddd*, which makes it possible to edit both numbers N and X by one instruction. To this end the numbers must occupy sequential fields of storage. Note, that the X field contains nine decimal digits and a sign, while the pattern for editing this field comprises only eight characters of digit selection. Therefore, the rightmost digit used only for rounding is not printed. The results of the program execution are listed in the form

N	SQRT(N)
1	1.000
2	1.414
3	1.732
...	...
100	10.000

**Example.** The previous example has shown both the techniques of handling decimal arithmetic instructions, and that the decimal arithmetic is inconvenient for in any way important calculations.

For the sake of comparison, consider a program for calculating  $Y = \sqrt{X}$  for floating-point single-precision numbers

1.	SQRTE	START	
2.		USING	*,15
3.		SAVE	(2)
4.		L	2,0(1)

---

5.		LE	0,0(2)
6.		LER	2,0
7.	B1	LER	4,2
8.		DER	4,0
9.		AER	4,0
10.		ME	4, =E'0.5'
11.		CER	4,0
12.		LER	0,4
13.		BNE	B1
14.		L	2,4(1)
15.		STE	0,0(2)
16.		RETURN	(2)
17.		END	

This program can be called with the aid of the macro instruction

CALL SQ RTE,(X, Y)

where X is the label of the word containing the argument, and Y is the label of the word for recording the values of square root.

Calculation of X is performed following the same scheme as in the previous example. It should be noted that the branch in statement 13 is performed on the condition code value formed in statement 11. We use here the fact that the condition code in the execution of the LER instruction does not change. Generally speaking, to write programs in this way is not recommended. However, this technique allows us to save one branch instruction as compared to the previous example.

Handling the subroutine parameters in statements 4, 5 and 14, 15 is known by the reader. In the Assembler language explicit addresses are commonly used in this case.

**Example.** Mathematical expectation, or mean value of a set of several numbers is determined as follows

$$M(x) = \frac{1}{N} \sum_{k=1}^N x_k$$

where  $N$  is the quantity of numbers. Let us write a program for reading a card deck (less than 1000 cards). Punched in columns 1-4 of the cards are positive integer numbers (in the form of 1234, 0025, etc.), calculations and prints  $N$  and  $M(X)$ . The mathematical expectation is calculated accurate to two places after the decimal point.

	START		
INFILE	DTFCD	DEVADDR = SYSIPT,	X
		IOAREA1 = CARD, EOFADDR=K3	
OUTFLE	DTFPR	DEVADDR = SYSLST,	X
		BKLSIZE-11,	
		IOAREA1 = LINE	
K1	BALR	9.0	
	USING	*,9	
	OPEN	INFILE, OUTFLE	
	ZAP	N,=P'0'	
	ZAP	M,=P'0'	
K2	GET	INFILE	
	PACK	NUM, CARD(4)	
	AP	M,NUM	
	AP	N, =P'1'	
	B	K2	
K3	MP	M, =P'100'	
	DP	M,N	
	MVC	LINE, PICT	
	ED	LINE, N	
	PUT	OUTFLE	
	CLOSE	INFILE, OUTFLE	
	SVC	14	
PICT	DC	X'40202020222021204B2020'	
CARD	DS	XL80	
LINE	DS	CL11	
NUM	DS	PL3	
N	DS	PL2	
M	DS	PL5	
	EN	K1	

The program is fairly simple and needs no comments. The only question to the reader: why is the length of the M field equal to 5 bytes, but not 4 or 6 bytes?

## CHAPTER 8

### INTRODUCTION TO PL/I

#### 8.1. Problem-Oriented Programming Languages

Along with significant advantages over programming in a machine language, the Assembler language like the other symbolic programming languages, has a series of essential disadvantages. These, first

of all, are the large length and cumbersome nature of symbolic programs resulting from the necessity to explicitly write each machine instruction. Certain relief is yielded by the standard subroutines and macro instructions. However, most of the algorithm the programmer, all the same, has to break up into elementary operations corresponding to each of the machine instructions by himself. Secondly, a symbolic program written for an actual machine cannot be executed on a machine of another design. It is not surprising, since the main part of any symbolic language is machine instructions, while the instruction systems of diverse computers do not coincide with one another. The symbolic languages are rigidly tied to the machines they are written for, for which reason they are often called *machine-oriented* programming languages.

Appearance of many computers of various types, an increase in the number of programmers and programs developed by them have made for the emergence of a new approach to writing programs, a problem-orientation of programming languages.

As it has been mentioned in the previous chapter, in building programs most effort is taken by the coding operations. The symbolic programming facilitates this tedious work, but little, without adding creative element to it at all. That made the best programmers search for ways to write programs in the form more convenient for people.

It is evident, that writing the required actions in the form of a formula, say,

$$Y = A + B/X$$

is far simpler and clearer than the sequence of the instructions needed to make calculations according to this formula:

LE	0, B
DE	0, X
AE	0, A
STE	0, Y

The formula and the sequence of instructions are equally exact instructions for calculation of Y. However, the computer is unfortunately incapable of perceiving instructions written in the form of formulas convenient for man, and we have to resolve them into instructions. The only way to get rid of the tedious work of program coding is to make the computer perform this work by itself. i.e. to write a program (translator) which will convert instructions written in the form convenient for people into machine instructions.

This way proved to be promising, and beginning from 1954 programming languages begun to appear which were closer to the natural human language used to describe problems. These languages offer the programmer suitable program writing means regardless of the machine on which the programs will be executed later. Hence, these

languages are called *problem-oriented* programming languages, or *high-level languages*.

Programs written in high-level languages are far easier to learn, to code, and to understand than symbolic programs. Being stated in English or mathematical terms, they seem more natural than the abbreviations and mnemonic terms used in assembler languages. They usually require less writing, since one source statement produces many machine instructions. They are easier to debug, since they require fewer specific details. As shown by experience, their writing and debugging time is 1/3rd to 1/5th of that for the symbolic programming technique, not to mention the time taken to study these languages, for their more natural use of words and numbers.

The reader ought to ask us rightfully in this case why use is made of the Assembler language. The matter is that the programs formed by the translators from the problem-oriented languages take as a rule more storage space and are slower than the manually-coded programs. Therefore, gaining in the speed of programming, we loose efficiency of the working programs. More than that, the high-level languages permit information input/output from and to I/O devices only of several types of the whole multitude comprised by the devices that can be attached to a modern computer.

The Assembler language is commonly used when a program needs nonstandard information input/output, and also when the efficiency of a high-level language program does not satisfy the user requirements.

It should be noted, that the modern compilers from high-level languages build up fairly good programs comparable in their efficiency with programs in the Assembler language written by programmers of average abilities. Experience in programming shows that it will pay to write most programs in high-level languages, while the use of the Assembler language is rather limited, though it is necessary in certain problems.

It would be impossible to present several hundreds of high-level languages designed at present and leave the reader with a concrete understanding of their different capabilities and relative merits. Still greater is the number of compilers for various machines. However, only a few of them have become known throughout the world.

The oldest well satisfactory high-level language was FORTRAN (FORmulae TRANslator). It was developed specifically for solving scientific and engineering problems which can be represented by a set of arithmetic formulas and logical conditions. The modern versions of FORTRAN are powerful well-developed languages suitable for various computations. Now FORTRAN compilers are available on almost all computers.

Designed principally for mathematical usage, FORTRAN has some limitations ranging from certain restrictions in the use and

writing of FORTRAN statements to the impossibility of effective control of storage allocation.

These drawbacks had been overcome in ALGOL-60 (ALGO<sup>r</sup>ithmic Language) developed in 50s. ALGOL compares favourably with FORTRAN because of stricter description of the language and absence of unjustified restrictions. Another advantage of ALGOL is its powerful apparatus of subroutines. Soon after its appearance ALGOL became undoubtedly the most widely written about programming language in existence, and now descriptions of algorithms are published only in ALGOL. Ideas underlying ALGOL proved to be so fruitful that it served as a basis for designing a large group of programming languages.

In 1968 a new programming language ALGOL-68 was introduced. This is further development of the ALGOL-60 principles. However, because of the complexity of its perception and creation of its compilers resulting from the novelty of the ideas laid in it, the ALGOL-68 did not meet with wide acceptance accorded by the original ALGOL.

The use of computers in business data processing led to the design of a problem-oriented language convenient for writing business problems. The previous programming languages, such as FORTRAN and ALGOL do not suit this purpose.

Input/output is a prime concern in business application programs along with processing data of diverse types, and in addition to input/output limitations FORTRAN and ALGOL handle only binary fixed- and floating-point numbers.

All this led in 1961 to publishing the description of COBOL (Common Business Oriented Language). An important and obvious language characteristic of COBOL is the English-like syntax of the language statements which allows description of any type data. Besides, the COBOL design emphasizes features for specification of the properties and structure of input/output files, which is required in business data processing problems. However, COBOL has its own disadvantages the main of which is inability to efficiently perform cumbersome calculations which especially tells on now, when more and more mathematical methods find their applications in business and economy.

Because there are many programming languages, a problem arises what languages best suit the writing of algorithms for a wide class of tasks, and, therefore, what the programmers should be taught. Various tendencies have been outlined in this field. A standpoint is popular that the software base of computers should be constituted by several thoroughly chosen programming languages each of which is most suitable for writing a relatively narrow class of algorithms. For writing complicated programs, the programmers divide them into separate parts and choose the most suitable language for each

part. After all the parts of the programs have been translated, they are assembled into a unified program in the internal language of the machine.

According to another standpoint, use should be made of one high-level language so universal that it can be used to describe any data and operations on them carried out in modern computers. This standpoint is also shared by IBM specialists, and on the initiative of IBM, a new programming language PL/I (Programming Language) was developed between 1963 and 1966.

PL/I was developed after other fairly perfect languages and absorbed all their best features. It resembles FORTRAN, ALGOL, COBOL and certain other languages. It allows solution of more diverse scientific problems than those solved by FORTRAN on the one hand, and provides business information processing facilities not less effective than those provided by COBOL. With the use of PL/I, no other programming languages are needed, except for the Assembler language for non-standard programs. Finally, PL/I permits the programmer to use all capabilities of modern computers and this makes it compare favourably with the other languages.

The designers of PL/I succeeded in avoiding many snags in their work. The language created by them is far from becoming a simple combination of the features of other languages. On the contrary, these features have been further developed in PL/I and are based on common principles. One more merit of the PL/I language is that the user may operate only those language facilities which are needed for his problem. If that is the case, he has not to know anything about the other language facilities. This property makes it fairly simple to be studied by beginners who almost at once can produce meaningful small training programs. At the same time, it remains a powerful means of describing programs of any complexity.

Like many other languages, PL/I has not avoided its growing pains. For a long period of time efficient compilers capable of realizing all the PL/I features could not be created, for which reason many programming specialists treated it with scepticism. At present these difficulties have been overcome, and PL/I gains ever widening field of application to meet the needs of scientific, engineering and commercial programmers.

The goal of this chapter is not to provide complete description of PL/I. This language is so comprehensive that several thick volumes could be written and still some things would be left unsaid. The elements of PL/I covered by this chapter will allow us to write programs of average complexity.

**Example:** Let us consider a program for printing a table containing square roots of numbers ranging from 1 to 100 that has been written above in the Assembler language. Even this example helps

the reader in appreciating the compactness and clearness of writing algorithms in PL/I compared with symbolic programming.

```
TSQRT:PROCEDURE OPTIONS (MAIN);
    PUT EDIT ('bbbbNbbbbSQRT(N)')(A);
    N = 1;
    NEXT:PUT EDIT (N, SQRT(N)) (SKIP,F(6,0),F(9,3)),
    N = N + 1;
    IF N < 101 THEN GOTO NEXT;
    END TSQRT
```

The first string is the heading of the program and names it TSQRT. The PUT EDIT statement is used to output information onto a standard printer in the specified format. Enclosed in the first parentheses are the data to be output, and in the other pair of parentheses, the format these data are to be printed in. In our case the A format means that symbolic data are printed. The  $N = 1$  statement assigns N the value of 1. The next statement has a label NEXT which is separated from the statement by a colon. The SQRT is a PL/I built-in function computing the square root, and the square root has not to be calculated in the program itself. The list of formats includes a word SKIP which means to start the data at the beginning of a new line. Format F is for printing decimal fixed-point numbers. The first digit in the format is the number of places allocated to write the number in the print string, and the other digit shows the number of digits after the decimal point. The statement  $N = N + 1$  increments N by 1. Note, that the equal sign (=) in PL/I has a meaning other than in the mathematics. The IF statement is used in a PL/I program when a test or decision is to be made between alternatives. The literal meaning of this program is as follows:

```
IF N < 101 THEN GO OVER TO NEXT.
```

The END statement tells the compiler about the termination of the TSQRT program. A semicolon is used in PL/I to separate statements from each other.

## 8.2. PL/I Character Set

The character set of PL/I consists of 60 characters which may be divided into three groups:

- extended alphabet of 29 characters consisting of English letters from A to Z and three characters  $\text{X}$ ,  $\text{\#}$  and  $\text{@}$  treated as letters;
- ten digital digits from 0 to 9;
- 21 special characters, namely:

Blank- (as previously, sometimes designated by the character *b*)

Equal or assignment symbol =

Plus sign +

Minus sign	—
Asterisk or multiply symbol	*
Slash or divide symbol	/
Left parenthesis	(
Right parenthesis	)
Comma	,
Period	.
Symbol	;
Percent symbol	%
Semicolon	;
Colon	:
'Less than' symbol	<
'Greater than' symbol	>
OR symbol	
AND symbol (ampersand character)	&
NOT symbol	¬
Break character	—
Question mark	?

At present, the question mark has no specific use in the language, though it is included in the 60-character set.

In addition to the above listed characters, any characters of the EBCDIC code may be used in comments to the program and character strings.

Special characters may be combined to create other symbols of two characters:

**	denotes exponentiation
	concatenation
> =	greater than or equal
< =	less than or equal
¬ =	not equal
¬ >	not greater than
¬ <	not less than
/*	beginning of a comment
*/	end of a comment
— >	indicator sign

Blanks are not permitted in such character combinations. The characters in them must follow each other immediately.

### 8.3. Writing a PL/I Program

As distinct from the Assembler language, a punched card is not a basic unit for writing a PL/I program. The PL/I compiler treats a program as a continuous sequence of symbols continued from one card to another. One PL/I statement may occupy any number of punched cards. Several statements may also be punched into one

card. Any number of blanks may be written everywhere where the language rules allow at least one blank.

In order to make punching easier, it is usual to write the PL/I program, though very free in writing, on coding sheets, like the symbolic programs. One line of the coding sheet contains 80 positions corresponding to 80 columns of a punched card, each of these positions being for one character. Column 1 of all lines must contain a blank. Columns 73 through 80 are not examined by the compiler and serve to write the name of the program, and/or the serial number of the card. Therefore, PL/I statements may be written in positions 2 through 72 of each line of the coding sheet, being randomly arranged within a line.

It is good style to write a program in a fine manner, starting each statement at a new line, leaving spaces to emphasize the logical structure of the program, etc. This helps in program reading and updating.

To provide explanations of the program text, comments may be embedded within a PL/I statement where blanks are allowed. Comments are ignored by the compiler, and have no effect on the results of compiler work, but are included in the program listing.

Comments in a PL/I program have the following form

*/\*any sequence of EBCDIC code characters\*/*

The pair of characters *\*/* should not appear within a comment, otherwise it will be recognized by the compiler as an end of the comment, and the text that follows would not be treated as its continuation.

## 8.4. PL/I Syntax

Like any other programming language, the PL/I language requires that program be built in compliance with certain rules which together form the *language syntax*. Each PL/I statement has its own rules for being written which are given together with the statement description. This section of the text will deal with the rules applying to all statements of the language.

Any PL/I statement is composed of words and operational characters, and terminated by a semicolon separating one statement from another. The semicolon serves only this purpose and cannot appear within a statement.

Regardless of their use, all PL/I words are called *identifiers*. The identifiers are divided into two groups: PL/I identifiers, or keywords which are used to designate language statements, or properties of program entities; and user's identifiers which, like the symbolic name in the Assembler language, are used by the programmer to designate data the operations are performed on.

The list of keywords used in each PL/I statement is strictly fixed, while the identifiers the programmer may compose at will, but in compliance with the following rules:

- An identifier may be from 1 to 31 alphameric characters, provided that the first character is alphabetic. Recall that in PL/I, the characters  $\circ$ ,  $\#$  and  $@$  are considered to be alphabetic;
- The break character ( $\_$ ) can be used within an identifier to improve readability by connecting its parts;
- The break character can be neither the first, nor the last character of the identifier;
- Blanks and the other special characters are not allowed within identifiers.

Some examples of valid identifiers are as follows

```
INTEGRAL
X18
#_OF_ROW
SPOT_DOG
 $\circ$ @348
CGI_86_07
```

Examples of invalid identifiers are

```
NUMBER_OF_PRIMES_LESS_THAN_100000
  (contains more than 31 characters)
AREATH(X) (contains special characters, parentheses)
IST_POINT (begins with a digit)
FIELDI_ (is started with a break character)
RED FLOWER (contains a blank)
```

Certain identifiers, such as names of program entry points, or file names are used both in the program itself and in the operating system. More severe restrictions are imposed on such identifiers. Namely, they must contain from 1 to 6 alphameric characters, providing that the first character is alphabetic. The break character is not allowed in these identifiers.

The PL/I compiler accepts the source program as a continuous flow of characters. Separate elements of text are recognized only because they are separated by delimiters. Therefore, we must write THEN GOTO NEXT, rather than THENGOTO NEXT, or THEN GOTONEXT. The only exception to this rule is in that the words GO TO may be written either separately, or in one word.

Used in PL/I as delimiters are blank, comma, semicolon, colon, parentheses, and other special characters, except the break character. Delimiters may not replace each other, and the PL/I syntax strictly defines what delimiter must be written in each actual place. Blanks or a blank may be additionally embedded anywhere in the program

where a delimiter is required. They do not affect the compiler work and just facilitate program reading.

In the examples that follow any type of writing is equally valid:

OPTIONS	(MAIN)	OPTIONS	(MAIN)
A*	2	A*2	A*2
NEXT: N = 1;		NEXT:	N = 1;

Like in the Assembler language, the statements to which a branch may be made must be labeled. The label is an identifier and must be separated from the statement by a colon. A statement may have any number of labels, or no label. In the latter case, it is called an unlabeled statement. The general view of a PL/I statement is as follows:

*label: label: . . . label: PL/I statement;*

Note, that in most cases there is no point in furnishing a statement with more than one label.

A PL/I program consists of statements which are combined into *procedure* and *conventional blocks*. This program organization resembles subroutines and program sections in the Assembler language. A program in the Assembler language has an entry point. Similarly, any program in PL/I must contain the main procedure with which the execution of the program begins. In addition to the main procedure, the PL/I program may include other blocks, though they are optional. However, at present we shall do nothing more than writing a program containing only the main procedure. Like in the example given in Sect. 8.1, the first statement of the main procedure should be written as follows:

*label: PROCEDURE OPTIONS (MAIN);*

The label found in this statement is not used in the branch statement. In this event it is used to name the program entry point. To mark the end of a block use is made of an END statement having the following format:

*END [label];*

If used, the label in the END statement must contain the label of the procedure it is ending (must coincide with the name of the program entry point). It is used solely to help in finding the end of the block, since the END statement may be used for other purposes as well. Like the case is with the other PL/I statements, the END statement must be marked, if a branch may be made to it in the program. Therefore, the common format of the END statement is

*label: label: label: . . . label: END [entry-point-name]*

### 8.5. Data

The ES EVM system of instructions provides operations on data of diverse types. In Assembler programming storage space must be allotted to each elementary datum, then working storage, to the intermediate results, and then the instruction performing the required operations on them should be written, selecting instructions of fixed- or floating-point arithmetic, decimal arithmetic, Boolean logic, etc. for the corresponding types of data. In doing so, one has to watch the position of the decimal point, perform the required conversions, if data of diverse types are encountered, etc. The PL/I compiler performs this tedious work by itself. It only must be told certain characteristics of the data encountered in the program (number of decimal digits and position of the decimal point for decimal data, precision of representation for floating-point data, length for character string, etc.), and then the compiler will place them in storage, select the needed machine instructions for execution of the operations, and perform the required conversions.

PL/I allows operations on arithmetic and string data. The arithmetic data include:

- binary fixed-point data,
- binary floating-point data,
- decimal fixed-point data,
- decimal floating-point data,
- decimal character data.

The string data are of two kinds: data of the character-string type and data of the bit-string type.

In order to be able to perform operations on data, the PL/I compiler must know certain characteristics of them. For the arithmetic data these characteristics are:

- base (decimal or binary),
- scale (fixed-point or floating-point),
- mode and precision (the number of digits and position of the point for fixed-point data, and the number of digits in the mantissa for floating-point data).

For the string data the only characteristic is the length (in characters or bits).

Assignment of characteristics to constants encountered in the program is carried out by the compiler itself, proceeding from the constant declaration. Thus, for instance, 123.45 is a decimal fixed-point constant containing five decimal digits of which two are after the decimal point; constant 'ABCD' is a character-string constant four characters in length.

Assignment of characteristics to variables encountered in the program must be made by the programmer explicitly. To this end use is made of the DECLARE statement.

No actions are performed in the program on a DECLARE statement. It serves only to supply information to the compiler. Its general view is as follows:

```
DECLARE identifier1 attributes,
       identifier2 attributes
       . . . identifiern attributes;
```

Identifiers appearing in the DECLARE statement serve to assign names to variables and their *attributes* are used by the compiler to assign the desired characteristics to the variables.

Data of the arithmetic type are represented by base attributes (DECIMAL or BINARY), scale attributes (FIXED — fixed-point, or FLOAT — floating-point), and mode and precision attribute.

To assign characteristics to decimal character data being decimal numbers stored in the form of character strings, there is a special attribute PICTURE.

String data are declared with an attribute CHARACTER or BIT, and a length attribute.

**Fixed-point decimal data.** Declaration of a fixed-point decimal variable has the form

```
DECLARE identifier DECIMAL FIXED (p, q);
```

The attributes may follow in any sequence, but the mode and precision attribute (*p*, *q*) may not be the first one; *p* stands for the total number of decimal digits, and *q* — for the number of digits after the decimal point. If a variable contains only integer values, i.e. *q* = 0, then the attribute may be written in the form (*p*). For example, a variable X having three digits after the decimal point and a value not in excess of 1000 can be declared as follows:

```
DECLARE X DECIMAL FIXED (6, 3);
```

The variable X may vary from -999.999 to +999.999.

The *fixed-point decimal constant* is written in the form of a sequence of decimal digits with an optional sign and optional point. Unsigned constants are regarded as positive. If there is no decimal point, the constant is considered to be an integer.

Given below are examples of writing constants with attributes assigned to them by the compiler:

+25.0	DECIMAL FIXED (3,1)
1276	DECIMAL FIXED (4,0)
-001276	DECIMAL FIXED (6,0)
+0.025	DECIMAL FIXED (4,3)
.025	DECIMAL FIXED (3,3)
273	DECIMAL FIXED (3,0)

In storage, the fixed-point decimal data are represented by packed decimal fields.

**The fixed-point binary data.** Declaration of a fixed-point binary variable has the form

DECLARE *identifier* BINARY FIXED (*p*);

The precision attribute specifies the number of bits in the binary number. Fixed-point binary numbers can be only integer and you need not to specify the position of point. For example, a fixed-point binary variable whose value is not greater than eight bits may be declared as follows:

DECLARE BN8 BINARY FIXED (8);

This variable may vary from  $-11111111_2 \leq \text{BN8} \leq 11111111_2$ , or in decimal representation form  $-255 \leq \text{BN8} \leq +255$ .

The *fixed-point binary constants* are written in the form of a sequence of binary digits (0 and 1), possibly with a sign immediately (without blank) followed by the letter B. Examples of writing fixed-point binary constants and attributes assigned to them by the compiler are given below:

		<i>Decimal Equivalent</i>
10110B	FIXED BINARY (5)	26
+101B	FIXED BINARY (3)	5
-01B	FIXED BINARY (2)	-1

Fixed-point binary data are stored in storage in the form of full-words.

**The floating-point decimal data.** Declaration of a floating-point decimal variable has the form:

DECLARE *identifier* DECIMAL FLOAT (*p*);

The precision attribute (*p*) defines the number of decimal digits in the mantissa of the variable. For instance, a floating-point decimal variable with five significant digits can be declared with the aid of the statement

DECLARE XD DECIMAL FLOAT (5);

As in the Assembler language, the *floating-point decimal constants* consist of two parts: a mantissa and an exponent. The mantissa is a fixed-point decimal number, and the exponent is represented by the letter E followed by an exponent in the form of one or two decimal digits which may be signed. For example:

<i>Constant</i>	<i>Attributes</i>	<i>Value</i>
20E7	FLOAT DECIMAL (2)	$2.0 \cdot 10^6$
-0.1E-3	FLOAT DECIMAL (2)	$-1.0 \cdot 10^{-4}$
.345E0	FLOAT DECIMAL (3)	$3.45 \cdot 10^{-1}$
+000447E18	FLOAT DECIMAL (6)	$4.47 \cdot 10^{20}$
-8.000000E-5	FLOAT DECIMAL (7)	$-8.0 \cdot 10^{-5}$

**The floating-point binary data.** Declaration of a floating-point binary data has the form

DECLARE *identifier* BINARY FLOAT (*p*);

The precision attribute (*p*) defines the number of binary digits (bits) in the mantissa. Floating-point binary constants are written in a way similar to the decimal constants, but the mantissa may contain only binary digits (0 and 1) and the constant is immediately followed by the letter B. The exponent in those constants is written in the form of a decimal number, but stands for power of 2 by which the mantissa is multiplied.

Regardless of the base of representation, floating-point data are represented in storage by floating-point binary numbers of single-precision or double-precision as dictated by the number of digits in the mantissa.

**Maximum value of precision attribute.** For each type of numbers, the compiler has a maximum value of the precision attribute determined by the internal representation of data in the machine memory. The corresponding values of the precision attribute are given below:

DECIMAL FIXED (15,q)  
 DECIMAL FLOAT (16)  
 BINARY FIXED (31)  
 BINARY FLOAT (53)

For fixed-point decimal data the number of fractional digits should not exceed the total number of digits in the number, i.e.  $q \leq p \leq 15$ .

**Default rules.** To write all attributes to each variable in the DECLARE statement is rather tedious. To avoid this, the PL/I compiler has a built-in *default value* for each attribute. This value is assigned to the attribute if it is not specified, but assumed. Those values are called *default attributes*.

If no base attribute in BINARY or DECIMAL representation is present, the datum is regarded as decimal: the default attribute is DECIMAL. If no scale attribute is present, the default attribute is FLOAT.

When no precision attribute is present it is assigned the following default values depending upon the other attributes:

DECIMAL	FIXED	(5,0)
DECIMAL	FLOAT	(6)
BINARY	FIXED	(15)
BINARY	FLOAT	(21)

If neither base attribute, nor scale attribute are present, then neither precision attribute may be specified, and in this case to define attributes by default, use is made of the first character of the variable identifier in a way shown below:

I, J, K, L, M, N	BINARY FIXED	(15)
Any other letter	DECIMAL FLOAT	(6)

Therefore, according to the default rules, it is enough to write down only those attributes of variables which differ from the standard default attributes that can be assigned by the compiler.

Another way of cutting down the amount of writing work consists in taking the attributes out of the parentheses; variables having common attribute are parenthesized, and the attribute is written only once following the right-hand parenthesis

**Example.** The following two forms of writing the DECLARE statement are absolutely legitimate:

```

DECLARE A DECIMAL FIXED (7,2),
        B BINARY   FIXED (20),
        C DECIMAL  FLOAT (6),
        D BINARY   FLOAT (53),
        K BINARY   FIXED (15),
        X DECIMAL  FLOAT (7),
        Y BINARY   FLOAT (21),
        Z DECIMAL  FIXED (5),
        ☐ BINARY   FIXED (15),
DECLARE A FIXED (7,2) X DECIMAL (7), D BINARY (53),
        (B FIXED (20), Y) BINARY,
        (Z, ☐ BINARY) FIXED;
```

The DECIMAL attribute is omitted for variables A and Z; with variable B, all attributes differ from the default attributes, and none of them may be omitted; the FLOAT attribute is omitted for variables D, X, Y; the DECIMAL attribute may neither be omitted for variable X, because at least one attribute must be used together with a precision attribute; variables Y, Z and ☐ have their precision attribute omitted, because it has a value equal to the default one; declaration of variables C and K is optional, for all their attributes coincide with the default attributes.

Two remarks should be made to the use of the attributes of arithmetic variables:

1. Beginners in programming should not be too enthusiastic about compacting the amount of writing, for this makes the program structure less clear and interfere with making amendments;

2. In most tasks, the default attributes assigned by the compiler will quite do, and the other attributes should be specified, only if necessary.

**Decimal character data.** These data are represented in storage in the form of decimal zoned format fields, and are declared in the DECLARE statement as follows:

DECLARE *identifier* PICTURE' *character-string-picture*';

The *character-string-picture* should be constructed so that the conversion of arithmetic data into a numeric character form is easy to be performed with the aid of the ED and EDMK editing instructions.

A number of characters may be used to describe numeric data in a picture. The basic picture specification characters include:

- 9 Specifies that the associated position of the picture may contain only a successive digit of the number;
- V Specifies the position of the implied decimal point. No decimal point actually appears in storage nor is there any byte reserved for the decimal point. The V merely enables the PL/I compiler to receive precision information;
- S Indicates that the sign of the value (+if  $> 0$  and -if  $< 0$ ) is to appear in a separate byte position in the variable (in the field). The S picture specification may appear to the left of all digit positions in the picture;
- + Indicates that a plus sign (+) is to appear in the character representation of the variable if the value is greater than, or equal to, zero and that a blank is to appear if the value is less than zero;
- Indicates that a minus sign (-) is to appear in the character representation of the variable if the value is less than 0 and that a blank is to appear if the value is greater than, or equal to, zero;
- E Indicates the beginning of the exponent for floating-point numbers. It is placed directly in this position;
- K The same as E, but occupies no place in the field and serves solely to supply information to the compiler;
- T Indicates that this byte of the field must contain the sign of number,  $C_{16}$  if the number is greater than, or equal to, zero, or  $D_{16}$  if the number is less than zero;
- I The same as T, but if the number is less than zero, the zone contains  $F_{16}$ ;

R The same as T, but if the number is greater than, or equal to, zero the zone contains  $F_{16}$ .

A V and one of the sign representation characters S + —TIR may not appear more than once in a picture, characters T, I, R being only to the right of all digit positions in the picture (the field). An exception is floating-point field whose mantissa and exponent may have their own signs.

### Example

<i>Number</i>	<i>Picture</i>	<i>Storage field</i> (character form)
4143	'S9999'	+4143
—12	'—99999'	—00012
—23.47	'+999V99'	b02347
345	'V999E99'	345E03
—41.2	'S9V999E9'	+4120E1
—0.0083	'S99V9KS99'	—830—06

### Example

<i>Number</i>	<i>Picture</i>	<i>Storage field</i> (hexadecimal form)
+734	'99T'	F7F3C4
—734	'99T'	F7F3D4
+734	'99I'	F7F3C4
—734	'99I'	F7F3F4
+734	'99R'	F7F3F4
—734	'99R'	F7F3D4

The picture elements indicated above are mainly used for exact description of data read from punched cards.

The character-string picture may contain other characters used mainly for editing data to be printed out:

Z Specifies digital positions in which nonsignificant zeros must be suppressed;

\* It is used in much the same way as Z, but zeros are suppressed by the character \*;

It is introduced into a specified position if at least one significant digit or character V has been encountered before, otherwise it is replaced with a blank or \* character. Note, the character . does not specify by itself the position of the decimal point in a number;

, The same as the period. A comma is inserted in the specified position;

B Indicates that a blank is inserted in a specified position.

The Z and \* characters may appear only in the left part of the character-string picture. The asterisk cannot appear with the picture character Z.

In order to place a sign directly before the first significant digit of a number, use may be made of the S, + and — signs, observing the following rules:

— These characters can be used as *static*, in which case they are specified only once in a picture specification and appear in the associated data item in the position where they have been specified;

— These characters can be also used as *drifting*, in which case they are specified more than once (as a string) in a picture specification and appear to the left of the significant portion of the data item (field). When drifting characters are present in the picture specification, a sign is inserted directly before the first significant digit of the number, or in the last position occupied by a drifting character. The drifting characters found to the left are replaced with blanks.

### Example

<i>Number</i>	<i>Picture</i>	<i>Storage field</i>
14.57	'S999V99'	+01457
14.57	'S999.99'	+000.14
14.57	'S999V.99'	+014.57
14.57	'SZZZV.99'	+b14.57
14.57	'S**9V.99'	+*14.57
14.57	'SSSSV.99'	b+14.57
+123	'———99'	bbb123
—123	'———99'	bb—123
14.57	'ZZ.ZZZV.ZZ'	bbb'14.57
1234	'ZZBZZZ'	b1b234
31081977	'99.99.9999'	31.08.1977
214.7	'ZZ.ZZ.ZZZZ'	bbbbbbb214

To compact writing in a picture specification, use may be made of a repetition factor which is a parenthesized unsigned decimal integer constant that specifies the number of occurrences of a picture specification character in a picture specification. For example, we may write '(8)9V99' in place of the picture specification '99999999V99'.

**String data.** Variables of the character-string or bit-string type are declared as follows:

```
DECLARE identifier CHARACTER (p);
DECLARE identifier BIT (p);
```

The length attribute (*p*) specifies the length in bytes or bits, respectively. In storage, character-strings occupy one byte per each character, and bit-strings, naturally, one binary digit position per

a bit. A bit-string always occupies one or several sequential bytes and begins with the zero bit of the leftmost byte of the field.

*Character-string constants* are sequences of any EBCDIC characters surrounded with symbol'

*'character-string'*

If one of the string characters is symbol', it must be written twice. If a character string consists of repeated portions, a repetition factor may be specified to write in a condensed form. In contrast to picture specifications, the repetition factor in this event is written to precede the first apostrophe and applies to the entire string rather than to a character. For example, the character-string constant for the string 'ABABABAB' could be written as (4)'AB'.

Bit-string constants are built on the same principles as the character-string constants do, but they may contain only digits 0 and 1. The end symbol ' must be followed by the character B. Examples are

'1011'B    '1110111'B    (56)'0'B

The parenthesized number preceding the last example is a repetition factor which specifies that the following bit or bits are to be repeated the specified number of times. The example shown would result in a string of 56 binary zeros.

## 8.6. Operations on Data

PL/I offers the programmer a large set of operations which may be performed on data of diverse types. There may be discriminated four types of operations:

- Arithmetic operations performed on data according to the rules of arithmetic;
- Logic operations are operations that follow the rules of symbolic logic (operations performed on bit strings);
- Comparison operations are examination of the relationship between two similar items of data;
- Concatenation operation is the only one that joints two strings in the order specified, thus forming one string whose length is equal to the sum of the lengths of the two strings.

It should be especially noted that, as a rule, the execution of any operation requires both operands to be represented in a similar form. Thus, to add two numbers one of which is in a floating-point binary format and the other, in a zoned decimal format, additional instructions must be inserted in the program to convert the numbers to a similar format. During programming in the Assembler language, these instructions must be written by the programmer himself. It is, therefore, natural always to strive to select such a way of data representation in which data conversions are not used, or used very little.

PL/I allows the programmer to avoid this. We may write  $A + B$ , for instance, without paying attention either to the notation, the form, or to the precision of the numbers named identifiers  $A$  and  $B$ , and the addition all the same will be performed correctly. If variables  $A$  and  $B$  have similar attributes, the compiler will simply form in the program an ADD instruction of the necessary type, otherwise the required conversion instructions will be inserted before addition.

This ability of PL/I is very convenient. At the same time, it is fraught with danger of neglecting, especially by beginners in programming, the importance of finding the best format of data representation. As a result, programs appear which contain many superfluous data conversions and are insufficient in work. Sometimes such programs run improperly, if the data conversion rules of PL/I are not in compliance with what is meant by the programmer.

Therefore, the above-mentioned programming rule which requires that the data representation formats be thoroughly considered and mixing of different data types in computations be avoided, if possible, should be followed in PL/I programming as well.

**Arithmetic operations.** The PL/I symbols for the five basic arithmetic operations are:

<i>Symbol</i>	<i>Operation</i>
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

If both operands have similar attributes, then to carry out the operation, the compiler forms generally-used instructions of binary arithmetic with fixed- or floating-point, or of decimal arithmetic, otherwise the operands are first converted in compliance with the following rules.

*Type.* The machine has no instructions to perform arithmetic operations on decimal fields in a zoned format, for which reason data described with a PICTURE attribute are always converted into coded arithmetic form.

*Scale.* If the operands differ in their scale, then the fixed-point operand is converted into the floating-point form. In doing so, the conversion result precision is taken so that the number representation precision is not reduced. For example, a datum with attributes

DECIMAL FIXED (8,4)

is converted into a datum with attributes

DECIMAL FLOAT (8)

*Base.* If the operands differ in their number system, then the decimal datum is converted into the binary number system. Exceptions are floating-point data. No conversion instructions are formed for them, for both the BINARY FLOAT data and DECIMAL FLOAT data are represented in storage by floating-point binary numbers.

*Precision.* If the operands differ only in their precision, no conversion occurs. How the result precision can be determined by the precision of the operands is shown below.

An exception to the above-mentioned rules is exponentiation, in so far as its operands, the base and superscript, are not equal in rights, as the case is in the other operations. The conversion of the operands for exponentiations is carried out according to the following rules.

1. If the base is in the floating-point form, while the superscript, in the fixed-point form with the number of fractional digits equal to zero, then no conversions are needed. In this case, the superscript is an integer, and the exponentiation is performed through multiplication.

2. If the superscript is a fractional number, it is converted into a floating-point form, and the computation is performed by the formula

$$x^y = e^{y \ln x}$$

3. If both operands are in the fixed-point form, but the superscript is a fractional number, then they are converted into the floating-point form.

4. If both operands are in the fixed-point form, the first operand is always converted into the floating-point form, except the case when the superscript is an integer unsigned constant.

5. If the precisions of the operands are such that the exponentiation result in a fixed-point form can exceed the permitted number of digit positions (15 decimal, 31 binary), then the base is converted to the floating-point form, though the superscript is an integer positive constant.

During execution of an exponentiation operation, the following special rules should be taken into account.

1. Raising a non-zero number to a zero power produces a zero in the result.

2. Raising a zero to a positive power produces zero.

3. Raising a zero to a zero or negative power is considered an error.

4. A negative number may be raised only to an integer power. The superscript must be in a fixed-point form with the number of fractional digits equal to zero.

5. Raising a negative number to a power of any other type is considered to be an error.

In execution of operations it is of importance to know the attributes of the result. With floating-point operands, the result precision is easy to determine. It is equal to the maximum value of the operand precision values. If after the conversion the operands are in a fixed-point form, the result precision is determined by special formulas in which use is made of the following designations:

- $p$  — number of digits in the result;
- $q$  — number of fractional digits in the result;
- $p_1$  — number of digits in the first operand;
- $q_1$  — number of fractional digits in the first operand;
- $p_2$  — number of digits in the second operand;
- $q_2$  — number of fractional digits in the second operand.

In the fixed-point addition and subtraction operations, the result precision is determined by the formulas

$$\begin{cases} p = 1 + \max(p_1 - q_1, p_2 - q_2) + \max(q_1, q_2) \\ q = \max(q_1, q_2) \end{cases}$$

In the fixed-point multiplication operation the result precision is determined by the formulas

$$\begin{cases} p = 1 + p_1 + p_2 \\ q = q_1 + q_2 \end{cases}$$

In the fixed-point division operation the result precision is determined by the formulas

$$\begin{cases} p = 15 \\ q = 15 - (p_1 - q_1 + q_2) \end{cases}$$

for decimal operands, and by the formulas

$$\begin{cases} p = 31 \\ q = 31 - (p_1 - q_1 + q_2) \end{cases}$$

for binary operands.

**Example.** Let the following variables be declared in a program

```
DECLARE A DECIMAL FIXED (6,2);
        B DECIMAL FIXED (4,3);
```

then

(a) the sum or difference of A and B may be declared with the attributes

```
        DECIMAL FIXED (8,3)
```

(b) the product A\*B will have the attributes

```
        DECIMAL FIXED (11,5)
```

(c) the quotient of A/B will have the attributes

DECIMAL FIXED (15,8)

(d) the product of 3.5\*A will have the attributes

DECIMAL FIXED (9,3)

for constant 3.5 has attributes DECIMAL FIXED (2,1).

**Logic operations.** Three logic operations are provided in PL/I:

<i>Symbol</i>	<i>Operation</i>
$\neg$	NOT
$ $	OR
$\&$	AND

The NOT operation is a prefix operation. It applies to one operand, a bit string. The result is a bit string of the same length as the source length. It simply yields a result of the opposite condition: if a bit is 1, result is 0; if a bit is 0, result is 1.

The OR and AND operations are infix operations carried out bit at a time on each two similar bits of both operands. If the operands are equal in length, the result is a bit-string of the same length. Otherwise, the operand of shorter length is expanded with zeros to match the length of the longer string, and the result length will equal the longer operand string.

### Example

(a) The result of the execution of operation  $\neg$ '01101'B equals '10010'B and has the attributes BIT (5).

(b) The result of the execution of operation '1010' B & '0110'B equals '0010'B and has the attributes BIT (4).

(c) The result of the execution of operation '001011' B '011' B equals '011011'B and has the attributes BIT (6).

**Comparison operations.** These operations are used to test (compare) two data items (operands) to determine the relationship that exists between them, and to determine whether the relationship is true or false. The eight allowed comparison operations are as follows:

<i>Symbol</i>	<i>Operation</i>
LT or $<$	Less than
LE or $< =$	Less than or equal to
$=$	Equal
GE or $> =$	Greater than or equal to
GT or $>$	Greater than
NE or $\neg =$	Not equal
NL or $\neg <$	Not less than
NG or $\neg >$	Not greater than

Note, that in fact there exist six comparison operations in all, for the operations  $\geq$  and  $\neg <$ ,  $\leq$  and  $\neg >$  are equivalent, and the programmer may utilize the form he has better accustomed to.

The result of a comparison operation is always a bit-string, one in length. If the relationship between the operands is true, the result of the comparison operation equals '1'B; if false, '0'B.

Depending upon the type of the operands being compared, there are three types of comparison:

algebraic,  
sign, and  
bit comparison

The algebraic comparison is made for numeric data by subtracting the operands. The comparison of sign and bit strings is a logical operation. If comparison is made of strings of unequal length, the shorter string of characters is padded with blanks on the right, and the shorter string of bits, with zeros.

If the operands of algebraic comparison have different attributes, then, first conversions are to be performed similar to those in a subtraction operation.

In certain cases PL/I allows data of diverse types to be compared. Thus, comparison may be made between a bit string and a character string. If that is the case, the bit string is first converted into a character string of equal length: a zero bit becomes set to 0, and a one bit, to 1. Arithmetic data may be also compared with a bit string. In this case, it is considered that the bit string contains an integer positive binary number having a precision equal to the string length. Bit strings may be used exactly in a similar way in arithmetic expressions, though it is unnecessary.

**Concatenation.** In PL/I there is a special operation that facilitates manipulation of string data. The operation is called concatenation. It means 'to join together' string (character or bit) data. It is specified by the operator `||`. For example, the execution of the operation

`'MAIN' || 'PROCEDURE'`

yields a single string 'MAINPROCEDURE' in which the operands follow each other with no blank. The length of the result is equal to the sum of the lengths of the operands.

In addition to character strings, operands of a concatenation operation may be bit strings and arithmetic data. The operation result is determined by the following rules:

1. If two bit strings are being concatenated, the result will be a bit string. The length of the result will equal the sum of the operand lengths.

2. When a character string is joined to a bit string, the latter is converted into a character string.

3. When a character string and an arithmetic datum are to be concatenated, the arithmetic datum is converted into a character sign. The result will be a character string.

4. Before concatenation of bit string and an arithmetic datum the latter is converted into a fixed-point binary form and then treated as a bit string. The result will be the same as in concatenation of two bit strings.

5. Concatenation of two arithmetic data is not permitted.

### 8.7. Expressions and Assignment Statements

The purpose of the assignment statement is to modify the values of variables encountered in a program.

The operator format is as follows

*[label:] variable = expression;*

While the statement is working, the value of the expression is computed, converted into a desired format, and transferred to the storage area assigned to the variable.

The equal sign (=) is referred to as the assignment symbol because it denotes the assignment statement. The assignment statement does not necessarily represent equality. The operation executed can be verbally stated as 'Assign the value of the expression on the right of the assignment symbol (=) to the variable on the left of the assignment symbol'. In comparison, the sequence order of the operands is of no importance, while that of the right and left terms of the assignment statement is essential. For example, the assignment statement

$$X = 1$$

moves the 1 constant to the storage field assigned to the X variable. However, to write

$$1 = X$$

is senseless, as a constant cannot be assigned a value.

The statement

$$X = Y = 5$$

is far from stating that the variables X and Y are to be assigned the value 5. To make it clear, let us single out individual elements of the assignment statement in compliance with its format

$$\begin{array}{l} X - \text{variable} \\ = - \text{assignment symbol} \\ Y = 5 - \text{expression} \end{array}$$

Now it is clear: a  $Y = 5$  comparison operation is under execution, and its result will be '1'B, if  $Y$  equals 5, otherwise the result will be '0'B. This very value is to be assigned to the  $X$  variable.

Note that it is better to write this statement in the form

$$X = (Y = 5);$$

This form of writing eliminates any ambiguity.

The concept of *expression* included in the assignment statement has been already encountered before. This is a combination of variables, constants, operation symbols, and parentheses written according to certain rules. Expressions in PL/I differ from the other expressions only in that they have a somewhat greater set of operations and a somewhat different order of expression computation.

If an expression includes more than one operation, the order of their execution is of importance. Thus

$$(A + B) * C \quad \text{and} \quad A + (B * C)$$

are absolutely diverse expressions. To explicitly specify the order of performing operations, use is made of parentheses.

Like in algebra, parentheses may not fully influence the order of performing operations. If that is the case, the order is defined by the priority of operations. In PL/I a series of seven priorities determines which operation will be performed first. These priorities are:

1. Prefix  $+$ , prefix  $-$ ,  $\neg$  (NOT),  $**$  (exponentiation)
2.  $*$  (multiplication),  $/$  (division)
3. Infix  $+$ , infix  $-$
4. Comparison operations  $<$ ,  $<=$ ,  $=$ ,  $>=$ ,  $>$ ,  $\neg<$ ,  $\neg=$ ,  $\neg>$
5.  $\&$  (AND)
6.  $|$  (OR)
7.  $\parallel$  (concatenation)

All operations of a higher priority must be performed prior to operations of a lower priority, if parentheses do not specify another order. Operations of similar priority are performed from left to right, except for the highest (first) priority operations which are performed from right to left. A distinction should be made between prefix and infix operations. If signs  $+$  and  $-$  appear between two operands, as the case is in the expression  $A - B$ , they refer to infix operations and have priority 3. However, in an expression of the form  $-A * B$ , the sign  $-$  defines a prefix operation of the highest priority.

**Example**

<i>Mathematical form</i>	<i>PL/I form</i>	<i>Explanation</i>
$-\sqrt[4]{X}$	<code>-X**.25</code>	Record <code>-X ** (1/4)</code> is invalid, for <code>1/4</code> in PL/I is an instruction to divide 1 by 4, rather than a fraction.
$\sqrt[4]{-X}$	<code>(-X)**.25</code>	
$\frac{A}{BC}$	<code>A/(B*C)</code>	Record <code>A/BC</code> is invalid, for the multiplication operation symbol is missed, and the compiler accepts <code>BC</code> as an identifier. Record <code>A/B*C</code> is neither valid. Operations of a similar priority are performed from left to right, and this expression means $\frac{A}{B} C$ . In this case, parentheses may not be dropped.
$2^{(3^X)}$	<code>2**3**X</code>	In the first example the parentheses may be dropped, for operations of a higher priority are performed from right to left; in the other example the parentheses must be used.
$(2^X)^3$	<code>(2**X)**3</code>	

Let us illustrate the sequence of operations and data conversions performed during execution of the assignment statement. Consider the following segment of a program:

```

DECLARE X BIT(2), Y DECIMAL FIXED (5,3),
        Z BINARY FIXED (7),
        AFLOAT (6); C CHARACTER (7);
1      Y = 3.5;
2      Z = Y**2;
3      A = Z + Y;
4      X = '10'B;
5      C = 'LINE' || Y + A < Z + X & X;

```

The digits in the left-hand column do not refer to the program. They are used to help in reference.

Statement 1 assigns the value of 3.5 to Y. No conversions are performed in this case, for the compiler prepares the constants in the desired form.

Statement 2 calculates `Y**2`. No conversion takes place in this operation. The expression is computed as `Y*Y`, and the result has the value 12.25 and attributes DECIMAL FIXED (11,6). As a next step, the very assignment is performed involving conversion of the result to a binary form and truncation of the fractional portion (Z is described as an integer). Finally, Z will be assigned the value of 12.

In statement 3, before the addition, Y is converted to a binary form, and the result equal to 15.5 is converted into a floating-point form.

Statement 5 is composed with a view to illustrating the rules for computation of expressions. Such statements are not used in programs. In order to trace the sequence of computations, it is good practice to use parentheses:

$$C = 'LINE' \parallel (((Y + A) < (Z + X)) \& X);$$

The reader is recommended to recall the priorities of operations to make sure that the actions must be performed exactly in the specified sequence.

Therefore, expressions within the innermost pair of parentheses are evaluated first. In computation of  $Y + A$  the augend is converted to a floating-point form. The result equal to 19 has attributes DECIMAL FLOAT (6). Then,  $Z + X$  is evaluated. The bit string representing X is converted into a binary number (equal to 2) and added to Z. The result equal to 14 is obtained in a fixed-point binary form.

In the execution of the comparison operation the operands must be again converted to a common format: number 14 is converted into a floating-point form. Relationship  $19 < 14$  is false, and the comparison result will be '0'B.

The remaining action is simpler: the result of the & operation will be a bit string '00'B which will be converted to a character form and linked (concatinated) to the character constant 'LINE'. Thus, the value of the expression in the right term of the assignment statement is equal to 'LINE00' and has the attributes CHARACTER (6). Since its length is less than C, a blank is added on the right in assigning to obtain a string of the desired length.

Hence, after the execution of this segment of the program, the variable C will be assigned the value 'LINE00b'.

As we can see from this small example, the working program will contain many additional instructions carrying out all above-mentioned conversions, though the PL/I program writing is fairly short.

We strongly recommend once more *to avoid the use of mixed data in computations.*

In conclusion of this section, consider the conversion rules utilized in the execution of the assignment statement.

1. If a variable assigned a value has a FLOAT attribute, then the evaluated expression is to be converted to a floating-point form. If the precision length of the expression is less than that of a variable, the mantissa is padded with the required number of zeros on the right; if it is greater than the precision length of the variable, unnecessary low-order digits of the mantissa are dropped.

2. If a variable assigned a value has a **FIXED** attribute, then the evaluated expression is to be converted to a fixed-point form and truncated or padded with zeros on the left or right to obtain the required precision length.

3. If character variable is assigned the value of a character string having a greater length, the excessive characters on the right are lost. If the string has a less length, then the required number of blanks is added to it on the right.

4. If a bit variable is assigned the value of a bit string that is longer than the bit variable, then the excessive bits on the right are dropped. If the string is shorter than the variable, then the required number of zero bits is added to it on the right.

### 8.8. Arrays and Operations on Them

In the previous chapters we have encountered sets of similar data known as arrays. It is convenient to arrange array elements in subsequent bytes of storage. To reference each of such elements, we had to calculate its displacement with regard to the beginning of the storage field occupied by the array.

Like any language of high level, PL/I includes facilities for working with arrays. These facilities allow us to describe arrays constituted by elements of any type and to easily reference either the whole array, or individual elements of the array.

Arrays used in a program as conventional variables must be declared in a **DECLARE** statement. To this end, there is a special means known as a *precision attribute*.

The bound attribute for one-dimensional arrays has the form  $(p)$ , where  $p$  is a positive integer defining the number of elements in an array. A bound attribute, if written, must immediately follow the array identifier, for example:

```
DECLARE VECTOR(50) DECIMAL FIXED(9);
```

This statement declares a **VECTOR** array comprised by 50 elements each of which is a fixed-point decimal integer, precision attribute 9. Now it is clear why a precision attribute may not be written immediately after a variable identifier: precision and bound attributes are written in a similar form, and the compiler tells them only from their location.

In addition to one-dimensional arrays, PL/I makes it possible to declare two-dimensional arrays used to represent, say, tables and matrices, three-dimensional arrays, etc. Note, that in practice we commonly deal with two-dimensional arrays.

For multidimensional arrays the bound attribute has the form

$$(p, q, \dots, t)$$

where  $p, q, \dots, t$  are positive integers defining the number of elements in each array dimension. For example, a matrix consisting of 10 rows and 20 columns can be declared as follows:

```
DECLARE MATRIX (10,20);
```

As with simple variables, omitted attributes of array elements may be defined by default. If that is the case, each of the 200 elements of the MATRIX array would have default attributes BINARY FIXED(15). (Why so, the reader is recommended to find it out by himself.)

To reference array elements use is made of *variable subscripts*. Like the bound attribute (the number of array elements), the subscripts are parenthesized to follow the array identifier (name), but stand for element numbers in each dimension. Thus, for example, to reference element 27 of the VECTOR array, it will do to write VECTOR(27). We reference the element found in row 3 and column 18 of the MATRIX array by writing MATRIX(3,18).

Therefore, in the PL/I programming we get rid of fairly tedious work of computing displacements of array elements (see Sect. 4.7 — a matrix transposition program). It is enough to specify the element subscripts, and the compiler will build up instructions computing the necessary displacements by itself.

Subscripts of array elements may be specified both by integers and by any arithmetic expressions as well. In the course of the program run, those expressions will be computed and converted into binary integer numbers. The only precaution should be taken to see that the result of subscript computation in compliance with the logic of your program is a positive number not greater than the associated array bound attribute specified in the DECLARE statement.

In the course of the program, no checks are made for subscript range condition, and reference to a location outside the bounds of the storage field occupied by the array as a result of invalid evaluation of subscripts is a widespread error which leads to unpredictable results sometimes of astonishing nature.

In the main storage, array elements are arranged in sequence one after another in subscript ascending order. With multidimensional arrays, the elements are so arranged that the rightmost subscript is changed first of all.

Let an array R(3,3) be declared in a program. Then its elements will be arranged in the program as follows:

```
R(1,1) R(1,2) R(1,3) R(2,1) R(2,2) R(2,3) R(3,1) (R(3,2) (R(3,3)
```

Like ordinary variables, the variable subscripts may participate in expressions and the assignment statement. For example, the following statement is quite legal:

```
MATRIX (N + 1,M - 1) = X**Y - VECTOR(N - 1);
```

However, in many programs, operations are encountered which are to be performed upon all elements of an array. For example, let it be necessary to assign the value 25 to all nine elements of array R in a program. We may write the nine assignment statements as follows:

```
R(1,1) = 25;
R(1,2) = 25;
R(1,3) = 25;
. . . . .
R(3,3) = 25;
```

but it is evident that this method cannot be realized commonly (imagine an array, say,  $100 \times 100$ , instead of  $3 \times 3$ ).

Another way of building a loop is as follows:

```
DO M = 1 TO 3;
DO N = 1 TO 3;
R (M, N) = 25;
END;
END;
```

The DO statement is described in detail in Sect. 8.10.

The loop technique is used in processing array elements in a such languages as FORTRAN and ALGOL-60. PL/I offers a new, very attractive feature of using an array in the assignment statement as a whole, i.e. to write simply

```
R = 25;
```

This means that all elements of the R array must be assigned the value of 25. Therefore, in PL/I the concept of the assignment statement is extended and arrays can be used in it. Naturally, this feature leads not only to the possibility of assigning similar values to all elements of an array.

With arrays the common format of the assignment statement is as follows:

$$\text{array-identifier} = \begin{cases} \text{scalar-to-array} \\ \text{array-to-array} \end{cases}$$

*Scalar-to-array* is a familiar expression consisting of operators, parentheses, constants, simple and subscripted variables. The value of the expression is calculated once. Then, it is assigned to all elements of the array found in the left-hand term.

If in addition to constants, simple or subscripted variables the expression includes array names (identifiers), the resultant expression is known as *array-to-array*. The arrays included in the expression and also the array in the left-hand term of the statement must have identical bounds (bound attributes).

The array assignment statement is equivalent to the sequence of ordinary assignment statements which contain similar elements of all arrays present in the statement. The array elements are assigned values element after element according to their arrangement in storage.

**Example.** Let arrays A, B, and C be declared with bounds (2,3), Then:

(a)  $A = B$ ; means that each element of array A is assigned the value of the corresponding element of array B.

(b)  $A = A + 3$ ; means that one and the same number 3 is added to each element of array A.

(c)  $A = B * C$ ; means that in execution of this statement the corresponding elements of arrays B and C having similar subscripts will be multiplied together. This statement is equivalent to six conventional assignment statements

$$\begin{aligned} A(1,1) &= B(1,1) * C(1,1) \\ A(1,2) &= B(1,2) * C(1,2); \\ &\dots\dots\dots \\ A(2,3) &= B(2,3) * C(2,3); \end{aligned}$$

Note that the execution of the statement  $A = B * C$  is not equivalent to multiplying matrices according to matrix algebra.

(d) The example that follows shows that certain precaution should be taken in use of array assignment statement.

Let the value of  $A(1,2)$  be added to each element of array B and the result be assigned to the corresponding element of array A.

To make it more understandable assign the elements of arrays A and B the initial values

$$A = \begin{pmatrix} 1 & 3 & 5 \\ 7 & 9 & 11 \end{pmatrix} \quad B = \begin{pmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \end{pmatrix}$$

then execution of the statement

$$A = B + A(1,2);$$

will yield the result

$$A = \begin{pmatrix} 5 & 5 & 5 \\ 7 & 7 & 7 \end{pmatrix}$$

To better understand this result invalid at first sight, let us write the equivalent sequence of conventional assignment statements:

$$\begin{aligned} A(1,1) &= B(1,1) + A(1,2); \\ A(1,2) &= B(1,2) + A(1,2); \\ A(1,3) &= B(1,3) + A(1,2); \\ &\dots\dots\dots \\ A(2,3) &= B(2,3) + A(1,2); \end{aligned}$$

Now it is clear, that the value of the element  $A(1,2)$  is changed during the execution of the statement

$$A = B + A(1,2);$$

and it cannot be used to solve the task set. The correct solution is

$$Z = A(1,2);$$

$$A = B + Z;$$

where  $Z$  is a simple variable declared with attributes similar to those of the elements of array  $A$ .

### 8.9. GOTO Statement. IF Statement

Generally, the statements in a program are executed following the sequence they are written in. To alter the natural order of executing statements, use is made of a GOTO statement. As it has been mentioned above, the word GOTO may be written either as one word, or separately (GO TO).

The GOTO (branch) statement has a simple format:

GOTO *label*;

The label utilized in a branch statement must be present in another statement of the program. Before going any further, it is worth while recalling that the label is to be built in compliance with all rules for writing identifiers. It precedes the statement and is separated from it by a colon (:). The statement may be unlabelled. If that is the case, no branch to it is possible. Any executable statement including the GOTO statement itself may be labelled. Non-executable statements of PL/I, such as, say, the DECLARE statement, may not be labelled. The GOTO statement performs an unconditional branch.

An IF statement is used in a PL/I program when a test or decision is to be made between alternatives.

The general format of the IF statement is as follows

[*label*]: IF *expression* THEN *statement-1*;  
                                  [ELSE *statement-2*];

Commonly, an 'expression' in the IF statement is comparison. In this event the statement is executed as follows:

— If the condition tested is true, we execute the statement following the THEN, while the statement following the ELSE is ignored. Then we execute the statement that follows the whole of the IF statement;

— If the condition tested is false, we perform the statement following the ELSE, while the statement following the THEN is ignored.

**Example**

(a) IF  $X > Y$  THEN  $X = X + Y$ ; ELSE  $Y = X + Y$ ;

If in execution of the IF statement, the  $X > Y$  condition is true, the  $X = X + Y$  statement will be executed. Otherwise, the  $Y = X + Y$  will be executed.

(b) If  $A \neq B$  THEN GOTO M1; ELSE GOTO M2;

If A is not equal to B, a branch will occur to the M1 label, otherwise, next to be executed will be the statement labelled M2.

The logic of many programs requires that certain actions are performed only if the condition tested is true, otherwise, no actions should be taken at all. In programming such branch operations use may be made of the IF statement in its short form (without ELSE).

For example:

IF  $X + Y > Z$  THEN GOTO BAD;

If the  $X + Y > Z$  condition is true, a branch to the BAD label will occur. If the condition tested is false, nothing will be done. The program proceeds to the next sequential statement following the IF statement.

Sometimes actions are required only if the condition tested is false, and no actions are taken when the condition tested is true. In this case, the THEN structure may incorporate the so-called 'empty' statement. For example, the statement

IF  $A = 0$  THEN; ELSE  $X = Y$ ;

states that if the  $A = 0$  condition is true, nothing should be done, and if it is false, the  $X = Y$  assignment is performed. Note right now, that the last statement may be written in a simpler way:

IF  $A \neq 0$  THEN  $X = Y$ ;

Not only can the IF statement contain simple conditions, but any expressions may be written in it. In this case, the expression is evaluated to obtain its value and, if necessary, is converted into a bit string. If the string contains even one bit other than zero, the program executes the statement following the THEN. If all bits of the string are set to zero, it performs the statement following the ELSE, or nothing is done, if the case is with a shortened IF statement.

**Example.** A branch to an LBL label must occur at a certain point of the program, if variables N1, N2, and N3 are set simultaneously to 1, 5, and 19, respectively. The branching is performed by the statement

IF  $N1 = 1 \ \& \ N2 = 5 \ \& \ N3 = 19$  THEN GOTO LBL;

**Example.** Let the point  $A$  have coordinates  $(X, Y)$ . If the point  $A$  fits the shaded area shown in Fig. 8.1, calculate the square of its distance from the coordinate origin point. The solution is by the following IF statement:

```
IF ((X >=1) & (X <=2) & (Y >=1) & (Y <=2)) |
  ((X >=2) & (X <=3) & (Y >=0) & (Y <=2))
  THEN R = X ** 2 + Y ** 2;
```

Note, that the parentheses are used here only to help clarify the statement. The reader is recommended to look through the section describing the order of operation execution to make sure that all parentheses may be dropped.

The above solution is not the best, but the simplest one. The IF statement may be written in a shorter way with dropping unnecessary parentheses:

```
IF X >=1 & Y >=0 & X <=3 & Y <=2 & (X >=2 | Y <=1)
  THEN R = X * X + Y * Y;
```

Thoroughly study both versions of the solution to make sure that only when the conditions tested are true, the point  $A$  fits the shaded area.

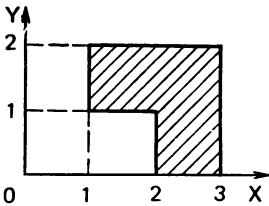


Fig. 8.1. Example of not simple condition in IF statement

In the format of the IF statement an IF statement may appear again as *statement-1* and *statement-2*. In this case, we speak of a *compound IF statement*.

**Example.** Let arithmetic variables  $A$ ,  $B$ , and  $C$  have unequal values being paired. Assign the variable  $X$  an average value of all of them. One of the possible solutions (most clear) is shown in Fig. 8.2.

The illustrated flowchart can be realized by one fairly large IF statement:

```
IF A < B THEN IF B < C THEN X = B;
  ELSE IF A < C THEN X = C;
  ELSE X = A;
ELSE IF A < C THEN X = A;
  ELSE IF B < C THEN X = C;
  ELSE X = B;
```

The reader is strictly recommended to do the following:

(a) Examine the flowchart to make sure that the result of its execution will be the average of the numbers  $A$ ,  $B$ , and  $C$  in the field  $X$ ;

(b) Trace against the flowchart the correspondence between the flowchart and the IF statement realizing it;

(c) Note the technique of writing the program: writing the sequential statements in a steplike manner and placing the ELSEs under corresponding THENs we add to clarifying the program and make

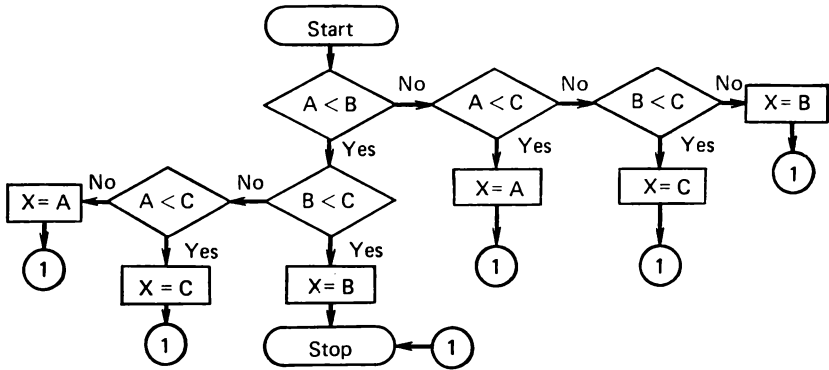


Fig. 8.2. Flowchart of computing average of three unequal numbers

it fine (which is of importance) to make the program reading easy and to avoid many errors.

This example is given here only to illustrate the compound IF statement.

The reader is asked to cut down the flowchart shown (naturally by thinking up your own more efficient algorithm), and to write the program segment realizing your flowchart as condensed as practicable.

When writing compound IF statements, take precautions in use of the ELSE clause. As an example, consider the following statement:

IF A < B THEN IF C < D THEN X = 0; ELSE Y = 0;

Its interpretation is ambiguous. As a matter of fact, to what does the ELSE Y = 0 refer — to the outermost or innermost IF? In order to avoid such an ambiguity, one should write down all ELSEs without failure, though some of them do not have to perform any action. In this case, the statement must be written in one of the following manners:

```

IF A < B THEN IF C < D THEN X = 0;
                    ELSE;
or
ELSE Y = 0;
IF A < B THEN IF C < D THEN X = 0;
                    ELSE Y = 0;
ELSE;
  
```

Take notice of the fact that these statements are used to perform different actions. The choice of the desired statement is dependent on the program logic.

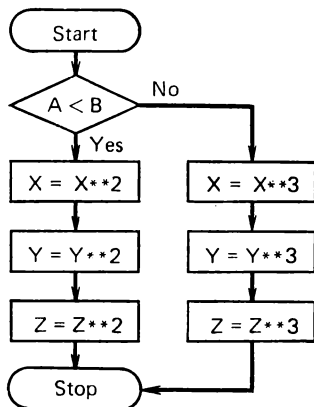


Fig. 8.3. Flowchart of program segment

### DO-group in an IF statement.

The IF statement is designed to execute one statement following the THEN or ELSE clause. Sometimes, however, it is necessary to execute more than one statement following the THEN or ELSE. This can be accomplished through the use of a DO-group. The DO-group is simply a series of PL/I statements headed by the word DO and terminated by the keyword END.

Let, to satisfy the condition  $A < B$ , the variables X, Y, Z be squared, otherwise they be

cubed. For the flowchart of this program segment, see Fig. 8.3.

By means of the PL/I facilities known to us, the program may be written as follows:

```

IF    A < B THEN GOTO L1;
      ELSE GOTO L2;
L1:   X = X ** 2;
      Y = Y ** 2;
      Z = Z ** 2;
      GOTO L3;
L2:   X = X ** 3;
      Y = Y ** 3;
      Z = Z ** 3;
L3:   . . . . .
  
```

It is clear, that this method is rather inconvenient. Using it, one has to utilize excessive labels and write additional GOTO statements. However, the IF statement format does not permit more than one statement to be specified after the THEN and ELSE.

As it has been mentioned above, to pass it over there are so-called DO-groups in PL/I. Any sequence of PL/I statements may be grouped into a DO-group. Such a DO-group may be used in the IF statement as a separate statement. With the aid of the DO-group our example may be written as follows:

```

IF A < B THEN DO;
    X = X**2;
    Y = Y**2;
    Z = Z**2;
END;
ELSE DO;
    X = X**3;
    Y = Y**3;
    Z = Z**3;
END;

```

### 8.10. DO-Loop

The logic structure of almost any program can be represented in the form of a set of various loops, and efficient organization of loops is one of the most important problems to be solved in programming.

In principle, the PL/I facilities that have been studied allow us to program any loop. However, PL/I incorporates a special DO-loop statement minimizing efforts in writing loops of whatever types.

The DO statement has several formats (types) intended for various loop end tests. These formats are considered below in the ascending order of their complexity.

**Format 1.** This format of the DO statement has already been explained in the previous section of this text;

```

DO;
    statement-1;
    statement-2;
    . . . . .
    statement-n;
END;

```

Therein the DO statement defines a DO-group rather than a loop. The statements found between the DO and END are executed once, no loop-end tests being performed. DO-groups are mainly used in IF statements. Distinctions between them and DO statements defining loops are described below.

**Format 2.** This type of the DO statement looks like the previous one. A distinction is in that it includes a variable known as the control variable:

```

DO    variable-expression;
      statement-1;
      . . . . .
      statement-n;
END;

```

The value of the expression is evaluated on the DO statement and assigned to the control variable. Then, the statements found between the DO and END are executed in sequence. The loop is run once with no loop-end test. As a matter of fact, this form of the DO statement does not organize a loop in the conventional form. The field of its application will be clear later.

**Format 3.** To organize iterative loops, it is good to utilize the DO WHILE statement. It has the following form:

```
DO    WHILE (expression);
      statement-1;
      . . . . .
      statement-n;
END;
```

At each run of the loop, the value of the expression is computed and is then converted, if necessary, into a bit string as does the condition tested in the IF statement. Till the string contains at least one bit other than zero (the condition tested is true), control will be transferred to the loop statements. As soon as the condition tested is false, i.e. contains only zero bits, control is passed to the program statement following the END.

The loop-end condition is tested before any statements in the DO-group are executed. Thus, it is possible that the statements following a DO WHILE and terminated with an END may never be executed. This would occur when the expression tested is false.

Let M and N be binary variables containing positive numbers. The program segment found below moves the value of the greatest common divisor of M and N into M and N. The calculations are made by the Euclidean algorithm:

```
DO WHILE (M  $\nabla$  N);
  IF M > N THEN M = M - N;
                N = N - M;
  ELSE N = N - M;
END;
```

Where possible, the authors strive to utilize, as examples, the algorithms which have been previously programmed by the other facilities. The reader is recommended to return to the previous text and thoroughly compare diverse approaches, paying special attention to the advantages of using languages of higher levels.

**Format 4.** The following variety of the DO statement is used for organizing loops with a count:

```

DO    variable-expression1 BY expression2 TO expression3;
      statement-1;
      . . . . .
      statement-n;
END;
```

The PL/I and TO keywords in this statement are used as follows: BY specifies the increment amount in an iterative DO statement, and TO specifies the limit value of the control variable.

At the beginning of the statement execution, the *expression<sub>1</sub>* is evaluated and its value is assigned to the *variable* (control variable). Next, *expression<sub>2</sub>* and *expression<sub>3</sub>* are evaluated, and one of the following loop-end tests is made:

```

if expression2 ≥ 0, test the condition
                      control variable > expression3;
if expression2 < 0, test the condition
                      control variable < expression3;
```

If the value of the control variable (*variable*) tested is true, the DO-loop is terminated, because the control variable is greater than the limit value (*expression<sub>3</sub>*). Otherwise, the loop statements are executed. Each time through the loop, the *modification value* (*expression<sub>2</sub>*) is added to the control variable, and control is passed to test the loop-end condition.

The program segment given below calculates the sum of squares of all even numbers not greater than 100:

```

N = 0;
DO;   K = 2 BY 2 TO 100;
      N = N + K*K;
END;
```

The execution of the program will yield the sum

$$N = 2^2 + 4^2 + \dots + 98^2 + 100^2$$

If we wish to summarize the numbers in the reverse order, from 100 to 2, we may specify a negative modification value, in which case a 'count down' operation is in effect:

```

N = 0;
DO K = 100 BY -2 TO 2;
  N = N + K*K;
END;
```

These two examples have introduced you to the complicated iterative loop-end test described above. If the modification value is positive, the loop is executed repeatedly until the value of the control variable becomes greater than the limit value specified in

the DO statement (TO clause). If the modification value is negative, the loop is terminated when the value of the control variable becomes less than the limit value.

The PL/I syntax provides a certain freedom in writing a DO statement for organizing loops with count.

1. The BY and TO clauses may be written in any order. Thus, the statements

```
DO K = 2 BY 2 TO 100;  
DO K = 2 TO 100 BY 2;
```

are absolutely similar.

2. Most often are loops with the modification value equal to 1. In this case, it is not necessary to specify the BY clause. If a DO statement contains no BY clause, the default modification value is equal to 1.

3. The TO clause may also be omitted in a loop statement. If that is the case, no comparison is made with a limit value (TO), and the termination of the loop must be organized by the programmer.

### Example

- (a) DO K = 1 BY 2 TO 100;
- (b) DO K = 1 BY 1 TO 100;
- (c) DO K = 1 TO 100;
- (d) DO K = 100 BY 1;

The loop in the example (a) will be executed for the following initial values of the control variable K: 1, 3, 5, . . . , 97, 99.

Examples (b) and (c) are two different forms of writing the same statement. In this loop, K goes through all integers from 1 to 100 inclusive.

No TO clause is used in example (d), and the loop will be executed for the K values equal to 100, 101, 102, . . . Note that it would be an error to omit the BY clause, too, in this statement though the modification value is 1, for the resultant statement could be executed only for K = 100 (format 2).

Very often use is made of iterative DO loops in which the control variable is used to subscript array elements.

**Example.** The program segment that follows calculates the sum of cubes of one-dimensional array elements:

```
DECLARE A(50) FLOAT, S FLOAT;  
S = 0;  
DO N = 1 TO 50;  
S = S + A(N)**3;  
END;
```

**Format 5.** This is a most general variety of the DO statement as if combining the two previous formats:

```
DO   control variable-expression1 BY expression2 TO expression3
    WHILE (expression4);
    statement-1;
    . . . . .
    statement-n;
END;
```

This loop is executed like the iterative DO. A distinction is in the loop-end testing. The DO statement of this format terminates the execution of the loop, if at least one of the following requirements is satisfied:

- The control variable has exceeded the limit value ( $expression_3$ );
- If the condition tested in the WHILE clause has become false.

The necessity of utilizing a DO statement of this format arises not often. However, sometimes it may be very convenient.

**Example.** Calculate the sum of cubes of one-dimensional array elements, the calculations being terminated as soon as a negative element is encountered in the array.

This problem is easy to solve using the following program

```
DECLARE (A(50), S) FLOAT;
S = 0;
DO   N = 1 TO 50 WHILE (A(N) >= 0);
    S = S + A(N)**3;
END;
```

As it follows from the above example, the DO statement of format 5 is used for iterative loops which are to be interrupted when a certain additional condition tested is true. The conventions on the rules of using the BY and TO clauses valid in this format are the same as in the previous format.

Let us examine the following example of the DO-loop statement

```
DO N = 1 WHILE (X > 0);
```

In this case, despite the presence of WHILE in the statement, the loop will be performed only once for  $N = 1$ . In order to assign the control variable  $N$  the whole numbers 1, 2, . . . , in the loop, a BY clause must be explicitly specified with the modification value of 1:

```
DO N = 1 BY 1 WHILE (X > 0);
```

The loop will be through, when the variable  $X$  becomes positive. The described varieties of the DO statement allow easy programming of almost all loop types that may be encountered in various

algorithms. There are, however, problems in which the loop organization facilities considered are not enough.

Let, for instance, a certain sequence of statements be repeated for the following values of the control variable K:

1, 2, 3, . . . , 9, 10, 20, 30, 40, . . . , 190, 200

Evidently, such an execution of a loop with the aid of iterations is impossible, as the modification value within it changes from 1 to 10. We can write two DO-loops in succession:

```
DO K = 1 BY 1 TO 10;
END;
DO K = 20 BY 10 TO 200;
END;
```

It is clear, however, that this solution of the problem is far from being efficient, as we have to specify twice the statements found between the DO and END, and there may be neither one nor two such statements.

The PL/I designers have allowed for this difficulty and implemented the following convention in the language: several clauses controlling the modification of the control variable may be written in one DO statement. They are also called *specifications*. A comma must separate specifications.

In our case, this DO loop statement may be as follows:

```
DO K = 1 BY 1 TO 10, 20 BY 10 TO 200;
  statement-1;
  . . . . .
  statement-n;
END;
```

The DO statements with several specifications are executed as follows: first the loop defined by the first specification in the list, as if there were no more specifications at all. Upon completion of this loop, control is not transferred, as the case generally is, to the statement following the END, and the loop specified by the next specification starts to be executed, etc.

No restrictions are imposed on the number of specifications in one DO statement. The following record, for instance, is quite tolerable:

```
DO N = 7, 19, 23, 25 BY 5 TO 50, 49 BY - 2 TO 1, WHILE (M = 0);
```

This loop will be executed once for N equal in sequence to 7, 19, 23, and then for all numbers divisible by 5 from 25 to 50, and then for all odd numbers from 49 to 1 (in the reverse order), and finally as many times as required until the variable M becomes

other than zero. During the execution of the last term of the loop, the variable N will remain to be set to the same value -1.

With DO statements containing several specifications, exercise much care in using the WHILE clause. Examine, for example, the following statements:

- (a) DO M = 1 TO 100 WHILE (Z < 0);
- (b) DO M = 1 TO 100, WHILE (Z < 0);

Though these statements are written almost in a similar manner, they are executed in absolutely different ways. The loop under the control of the (a) statement is executed for  $M = 1, 2, \dots$ , until the M becomes greater than 100, if the  $Z < 0$  condition does not become false before. The (b) loop is executed for  $M = 1, 2, \dots, 100$  at all times, and then at  $M = 101$  as many times as it is required to make the Z variable other than negative.

To conclude this section, we consider certain properties of the DO statement which must be known for the competent organization of PL/I loops.

**Attributes of the control variable.** In all the previous examples we have used integer variables as the control variable. The control variable, however, may be a variable with any attributes, say, a floating-point variable. All this equally applies to any variables, but when using control variables other than integers, the number of loop iterations must be thoroughly checked.

Let us consider the following segment of a program:

```
DECLARE X FLOAT;
DO X = 0.1 BY 0.1 TO 2;
. . . . .
END;
```

This program segment can compute, say, a table of values of a certain function with an argument varying from 0.1 to 2 at a modification value of 0.1.

If you do not know what the matter is in this case, then even prolonged speculation will hardly help you in finding an error in these three statements. The first time through the loop, the X variable will be equal to 0.1. The next times through the loop, the X will be set in sequence to 0.2, 0.3, etc., until it exceeds the limit value of 2. Therefore, the loop will be executed at the X varying from 0.1 to 2 at the modification value of 0.1. This looks as if what we need. Where on earth is the error?

Errors of this type are programmer's 'damned' errors. You may intently look through your program till irritation in your eyes, but everything looks correct and perfect, while the program stubbornly fails again and again!

The thing is that the programs are written on paper, while they are executed in real computers, and we must know how a particular operation is executed in a machine. The secret of our example is that, in contrast to whole numbers, floating-point numbers are represented in a machine not absolutely precisely. If you write 0.1 in a program, the number recorded in the corresponding storage location will be somewhat less, or greater, than 0.1. In conventional computations these inaccuracies are inessential, in so far as any computations themselves are performed with a certain inaccuracy. However, in our example this inaccuracy becomes decisive. If the constant 0.1 is represented in storage by a number somewhat greater than 0.1, the variable X will gain a value somewhat greater than 0.1, then somewhat greater than 0.2, . . . , and finally a bit greater than 2. However, the loop does terminate when the  $X > 2$  condition is true! Therefore, the last time, for  $X \approx 2$ , the loop will not be executed. At the same time, if the constant 0.1 is represented in storage by a somewhat smaller number, the program will run properly.

To avoid such cases as these, keep in mind the following rule: *do not specify exact limit values in DO-loops with fractional control variables*. In our example a correct DO statement may be written as follows:

DO X = 0.1 BY 0.1 TO 2.05;

This loop will be without failure executed at  $X \approx 2$  and will not be executed at  $X \approx 2.1$ , for inaccuracy of representing numbers in a machine is far less than 0.05.

Note in passing that inaccuracy in representing floating-point numbers makes meaningless the condition tests of the  $X = Y$  type in which both variables are declared with a FLOAT attribute. This comparison may become true only by chance, since either X or Y have approximate values. In order to ascertain the fact that two floating-point values match each other, a relationship of the  $|X - Y| < \epsilon$  type must be tested, where  $\epsilon$  is a prescribed accuracy to which the computations should be made.

In addition to arithmetic variables, any variables may be used as loop control variables. Thus, we may write

DO NAME-'JACK', 'JOHN', 'JAMES';

Obviously, that specifications in the loops of this type may be only a list of the values utilized by the control variable of the DO loop.

**Branches.** To perform a branch to any program statement, it is enough to label this statement, and then to write a GOTO statement anywhere in the program. Should the statement to which control must be transferred, be contained between the DO and

its corresponding END, i.e. in the loop body, some additional rules must be observed.

A branch to any statement included in the DO-group (format 1) may be accomplished with no restrictions. It should be remembered only that after the END statement used in the DO-group, control will be transferred in a manner similar to the natural order of statements execution.

Let a program have the following segment:

```

IF X < Y
  THEN DO:
    L2: statement-1;
    . . . . .
    statement-n;
  END;
ELSE statement-m;
X = X + 1;

```

then GOTO L2 may be written anywhere in the program. In this case, however, control will be passed after the END statement to the  $X = X + 1$  assignment statement, and the *statement-m* will be omitted. The procedure is exactly the same, as if the DO-group has received control after execution of an IF statement. It is just the difference in the rules for performing branches that tells DO-groups from loops organized by DO-statements of other formats.

If the loop is organized with the help of a DO statement, control transfer to the loop body statements from outside is disabled. Thus, the sequence of statements that follows is invalid:

```

GOTO L47;
. . . . .
DO K = 1 TO 200;
L47: X = Y + Z;
. . . . .
END;

```

The only method of entering a DO-loop is by control transfer to the loop heading, for example:

```

CYCLE: DO N = 1 TO 25;
. . . . .
END;
. . . . .
GOTO CYCLE;

```

At the same time, there are no restrictions at all to exit from a loop, if its execution must be interrupted before its normal termi-

nation. Any branches in the loop body are also allowed. The example that follows shows permissible control transfer:

```
DO KT = 1 TO N;
    IF X > Y THEN GOTO P;
Q:LAMBDA = 0.2;
    . . . . .
GOTO Q;
    . . . . .
END;
    . . . . .
P: . . . . .
```

Let us dwell upon a widespread error. Often, it is necessary to interrupt a certain through run of the loop, and to branch to the next through run. Let there be, say, an array T (20) and its elements are to be given certain processing. In that, only positive elements are to be considered.

The required actions can be performed by the following sequence of PL/I statements:

```
DO K = 1 TO 20;
    IF T(K) <= 0 GOTO FINAL;
    . . . . .
FINAL: END;
```

It would be an error to transfer control to the DO statement rather than to the END statement, for in the latter case, the loop would start with the  $K = 1$ , and the program would 'get caught' in an endless loop almost for sure.

**Nested DO-groups.** DO-groups may be nested within other DO-groups. This makes it possible to organize, in a program loops, any nesting—two-fold, three-fold, etc. In case of nested DO-statements, each of them must have its own END. Each inner DO must be completely contained within the body of the outer DO. Partial overlap of loops, as shown in the example below, is not allowed:

```
S: DO M = 1 TO 100;
    . . . . .
T: DO N = 1 TO 100;
    . . . . .
    END S;
    . . . . .
    END T;
```

The following correct versions are possible:

```
S: DO M = 1 TO 100;
    .....
    END S;
T: DO N = 1 TO 100;
    .....
    END T;
```

These are sequential independent loops

```
S: DO M = 1 TO 100;
    .....
T: DO N = 1 TO 100;
    .....
    END T;
    .....
    END S;
```

These are nested loops. As is shown in the above examples, a DO-statement may be labeled and the label may be specified in the corresponding END statement. This feature of PL/I makes it possible to make the logic structure of the program more clear, and it is good practice to use it. Presence or absence of labels in the END statements have no effects on the execution of the program.

As compared with usual loops, programming of nested DO-groups in PL/I involves nothing new. To illustrate this, choice is made of a double loop transposing a square matrix, size  $N \times N$ .

This problem has been already programmed before in machine codes, and the reader, remembering how much work was consumed by computation of various displacements, will appreciate the PL/I segment below:

```
DO I = 1 TO N - 1;
  DO J = 1 + 1 TO N;
    R = A (I, J);
    A(I, J) = A(J, I);
    A(J, I) = R;
  END;
END;
```

It speaks for itself!

### 8.11. Edit-Directed I/O

Input of source data and output of computation results are needed in any program, and, naturally, the PL/I designers do not fail to pay their attention to this important aspect of programming. The user is offered a wide variety of features to organize his data

in a most convenient way and select a proper method of access to them. However, from the standpoint of teaching this variety of I/O features is a disadvantage rather than an advantage, for, to describe all I/O facilities available in PL/I rather fully will take far more space than this text can provide.

This section deals with the edit-directed I/O facilities. They provide convenient representation of source data, say, punched into cards, and also obtaining a printed document which contains computation results in any document form.

As to its structure, the edit-directed input/output corresponds to the formatted input/output of the algorithmic language FORTRAN, but is a bit more convenient.

The ES EVM programming system requires that the information on the external media be organized into files, i.e. data sets in the form accepted by the system. Any file used by the program must be described. To this end, the Disk Operating System Assembler language uses DTF (define the file) macro instructions which are employed by the program to build DTF tables. In PL/I, any file can be declared in the DECLARE statement as a variable, and the PL/I compiler will construct the corresponding DTF tables proceeding from the attributes of the file declared.

We shall not touch on the problem of describing files. Instead we shall take advantage of the fact that PL/I has two standard built-in files used for input/output operations encountered most often. These files need not be explicitly described in the DECLARE statement, since the PL/I compiler knows by default all their attributes, and it can construct the corresponding tables by itself.

For the input of source information, use can be made of the standard input file SYSIN. The records of this file are unblocked, 80 bytes long, i.e. correspond to the format of cards.

To dump the computation results to a printer, use can be made of the standard output file SYSPRINT. The records of this file are 120 bytes long and correspond to the print line.

The input/output statements must contain the name of the file involved in the information interchange. If the file name is omitted, the compiler substitutes SYSIN by default for the input operations and SYSPRINT for the output operations. The standard files neither need to be opened and closed. They are opened automatically prior to the execution of the first input/output operation, and closed by the operating system upon completion of the program work.

The edit-directed input/output involves additional processing of information to transform the data for the required method of representation. Therefore, along with the *data list* containing the data to be input or output, the edit-directed input/output statement includes a *format list* indicating that the data are represented on an

external medium. Representation of data in storage is determined only by the attributes of the variables in the data list.

The form of the edit-directed input statement is as follows:

```
[label:] GET EDIT (data-list) (format-list)
                [(data-list) (format-list)] . . . . .
```

The keyword GET defines the input statement, and the keyword EDIT points to that the input is edit-directed. Next, the statement contains a list of variables which will be assigned values read from punched cards. The variable list is parenthesized. The format list defines representation of each variable on an external medium. The pair (data-list) (format-list) may be repeated in the GET EDIT statement several times, but this commonly brings no advantages, for all variables can be simply included in one list and compared with the format list.

The form of the edit-directed output statement is as follows:

```
[label:] PUT [control-item]
                EDIT (data-list) (format-list)
                [(data-list) (format-list)] . . . . ;
```

In addition to the data and formats, the PUT EDIT statement may specify the print mode defining the arrangement of the output data in the printed document. The following control items are valid:

```
SKIP [(n)]
PAGE
LINE (n)
PAGE LINE (n)
```

where  $n$  is an expression which will be computed before the execution of the PUT EDIT and converted to the integer type.

The SKIP control item causes skip of  $n - 1$  blank line and means to start data printing on the next line. For example, if written is SKIP (2), the printer will print the line formed by the preceding statement PUT, then the paper will be pulled two lines (i.e. one line will remain blank), and the next sequential line will be started to form on the given PUT statement. If the (n) clause is omitted, i.e. SKIP is written, then default SKIP (1) is specified. Specifying SKIP (0), data may be reprinted on the same line. This trick is used, say, to underline or print in bold letters a heading or the like.

The PAGE control item causes a skip to the first print line of the next page of the document being printed.

The number of lines per page for the standard file SYSPRINT is default equal to 60. Another page size may be specified too, but

to this end, the file must be explicitly declared in a DECLARE statement.

The LINE format control item specifies that the next data item is to be printed on the  $n$ th line of a page. If  $n$  is greater than the current line number, the required number of lines is skipped to leave them blank. If  $n$  is greater than the page size in lines, the printing is started on the first line of the next page. If  $n$  is less than, or equal to, the current line number, a branch is made to a new page, and then the required number of blank lines is skipped.

The PAGE LINE control item comprises the above two items. A branch is made to a new page and then the paper is fed to the specified line.

If one of the control items is specified in the PUT statement, the clause EDIT (*data-list*) (*format-list*) may be omitted. This output statement is used only to set the print file to the required position. For example, the PUT PAGE statement causes paper skip to the beginning of the next page.

The GET and PUT statements refer to the stream-oriented input/output. From the logic standpoint the data on an external medium are considered to be in the form of a continuous stream of characters. The record boundaries are not marked. Going over from a card to another and skip from a line to the next one are performed automatically, after processing the required number of characters. To print information on a new line before the previous line is completed, specify explicitly the control item in the PUT statement. The way of going over to a new punched card is shown below.

**The data list.** The data list of input/output statement contains the variables which must be assigned the values to be entered or the values that must be printed at the output. The list elements are separated by commas, and the whole of the list is parenthesized;

```
DECLARE ABC FIXED BINARY (31),  
        XYZ FLOAT DECIMAL (16),  
        SYM CHARACTER (10);  
GET EDIT (ABC, XYZ, SYM) .
```

As a result of executing the GET EDIT statement, three groups of characters will be selected from the input stream of the SYSIN file. The length of each group is determined by the corresponding format (we do not yet specify the format). Next, the first two groups of characters, which must contain decimal numbers written according to the PL/I constant writing rules, are converted from the character form into a code; then they are reduced to the attributes of the variables ABC and XYZ. The SYM variable has an attribute CHARACTER for which reason the characters of group 3 are not converted, and their value considered as a string of characters is assigned to the SYM variable.

Not only simple variables must be specified in the data list, but also subscripted variables. Thus, after the execution of the statement

```
GET EDIT (K, A (4), A (12)). . .
```

the values of the simple variable K will be assigned to elements 4 and 12 of the array A.

It is clear, however, that in case of a large amount of input information, it is fairly tedious to specify each variable separately in the data list. Therefore, PL/I allows array identifiers in the GET and PUT statements to be written without subscripts. This means that all elements of the array are to be input or output. For example, after the execution of the statement

```
DECLARE H (40) FLOAT;  
GET EDIT (H) . . .
```

the program will enter and assign values to all 40 elements of the array H. Remember only that, first, all the 40 values must be punched and that, second, such a record stands for 40 simple data, rather than one datum, so that 40 formats must be specified for them. This is not difficult, because the language has a means which helps in shortening records in the format list.

The array elements are input or output in the same sequence as they are arranged in storage.

All the above equally applies to both statements, GET and PUT. However, it is permitted, in the output PUT EDIT statement, to additionally specify constants and expressions in the data list. The values of the constants are transformed in compliance with the format and printed. The values of the expressions are first evaluated and then transformed and printed. For example, during the execution of the statement

```
PUT SKIP EDIT ('FUNCTION ROOT', (A + B) * 0.5) . . .
```

the text FUNCTION ROOT will be printed on a new line, and then the value of the expression  $(A + B) * 0.5$  will be evaluated and printed.

Recall once more that if a current line is not completely filled after the execution of a PUT statement, the next PUT statement will add data to the same line. If you wish to print data on a new line, the control item must be explicitly specified.

**The DO clause in the data list.** Suppose that two arrays are declared in a program

```
DECLARE (A (10), B (10)) FIXED;
```

and it is necessary to input the values of these elements. If the values are punched in the following sequence

```
A (1), A (2), . . . , A (10), B (1), . . . , B (10)
```

then this operation can be executed after writing

```
GET EDIT (A, B) . . .
```

Suppose, however, that the data are punched in the following sequence:

```
A (1), B (1), A (2), B (2), . . . , A (10), B (10)
```

Now the solution is not so simple. One of the possible solutions is in programming a loop:

```
DO K = 1 TO 10;
  GET EDIT (A(K), B(K)) (format-list)
END;
```

PL/I provides one more facility of writing similar operations: the loop statement may be placed directly in the data list, i.e. to write

```
GET EDIT ((A (K), B (K) DO K = 1 TO 10)) . . .
```

This form of record has the following advantages: first, this shortens the record, and the program becomes more understandable and, second, the compiler can build a more efficient operating program. Therefore, this record of the data list should be used always when possible.

Take notice of the two sets of parentheses used to mark the data list boundaries. One pair of parentheses is required by the format of the GET statement, while the other marks the data the input of which is repeated by the DO statement. The matter is that though the DO statement may be inserted in the data list, its corresponding END statement is not allowed. Therefore, to mark the area in which the DO statement acts, use is made of parentheses, and the loop heading is written at the end of the loop.

All possible specifications of a loop are allowed in a DO statement built in an input/output statement.

The statement

```
DECLARE A (20) FLOAT;
GET EDIT ((A (N) DO N = 1 TO 5, 16 TO 20)). . .
```

inputs the values of the first five and the last five elements of the A array. The values of the other elements do not change. This example shows that one statement may be used to assign values to a part of the array elements.

The statement

```
DECLARE R (25, 25)
GET EDIT ((( R (I, J) DO I = 1 TO 25)
            DO J = 1 TO 25)). . .
```

organizes the input of the array elements by columns in contrast to the conventional practice of entering a two-dimensional array by rows, i.e. when the rightmost subscript is modified first of all.

In this case three pairs of parentheses are needed. The outermost pair encloses the whole of the data list, the next pair of parentheses marks the J loop area of action, and the innermost pair shows the nested I loop area of action.

The above statement is equivalent to the following program:

```
DO J = 1 TO 25;
  DO I = 1 TO 25;
    GET EDIT (R(I, J)) (format);
  END;
END;
```

Note, that should it be necessary to input the elements of the R array by rows, i.e. in the sequence

R (1, 1), R (1, 2), ..., R (1, 25), R (2, 1), R (2, 2), .... R (25, 25),

it would be most simple to write the statement as follows

```
GET EDIT (R) (format-list);
```

for the array elements are arranged in storage exactly in this sequence.

The statement

```
GET EDIT (N, (X, (M) DO M = 1 TO N)). . .
```

states that first the value of N is input which is then utilized as a count for the input of the values of the first N elements of the X array.

Let an array MATRIX (12,12) be specified in a program and its diagonal elements are to be printed. This function can be performed by the statement

```
PUT EDIT ((MATRIX (K, K) DO K = 1 TO 12)). . .
```

**The elements of the format list.** In case of data input, each element of the format list indicates the number of the input stream characters which contain the value assigned to the current variable from the data list. In addition to the input field width, the format contains information about data representation on an external medium. When a value associated with each format is output from the data list, it is converted into a character form and transferred into the output stream.

The edit-directed input/output allows representation, on an external medium, of all data types available in PL/I, except for fixed- and floating-point binary data punched and printed, which are represented by decimal numbers. This is natural, because the input

and output data of programs are worked on by the personnel accustomed to decimal representation, and the possibility of representing numbers on external media in a binary form is simply unnecessary.

There are formats in PL/I which correspond to all data types:

F( $w$ , [ $d$ , $p$ ]])	Fixed-point format
E( $w$ , $d$ [, $s$ ])	Floating-point format
P ' <i>picture-specification</i> '	Picture format
A [ $(w)$ ]	Character string format
B [ $(w)$ ]	Bit format

In this list the letter  $w$  stands for the width of a field occupied by the data on an external medium;  $d$  in the F-format indicates the position of the decimal point (number of digits after the decimal point), and in the E-format—the number of digits from the right of the fraction; the letter  $p$  in the F-format is a scaling factor; designation  $s$  in the E-format may be used to indicate the total number of digits in the mantissa.

Note, that the format type must not necessarily match the type of the variable assigned a value. Thus, for example, floating-point data may be input and output in format F, any arithmetic data, in format P, etc. Recall that the format defines only data representation on an external medium, though discrepancies between the format type and variable attributes may involve additional conversions affecting the efficiency of the operating program.

*F-fixed-point format.* An element of the F format having the form

$$F(w [, d [, p]])$$

states that the  $w$  current characters of the stream are interpreted as fixed-point decimal numbers. The number must be punched according to the rules for recording decimal fixed-point constants:

$$[sign] [digit \dots] [\cdot] [digit \dots]$$

In addition to the decimal digits, the number may contain an optional sign and an optional decimal point. If a number record takes less than  $w$  positions including separate positions for a sign or a decimal point, if present, the number may be located anywhere within the field of width  $w$ . Therefore, the number may be preceded or followed by an optional number of blanks to fill the field width to  $w$  characters. If the field contains only blanks, a zero value is introduced.

Designation  $d$  in the format indicates the position of the decimal point. If no decimal point is punched, it is assumed that there are  $d$  decimal places to the right of the field. If a decimal point is actually punched, its position overrides the  $d$  specification, and the number of digits after the decimal point is defined by its explicit position.

**Example**

<i>String of characters in input stream</i>	<i>Format specification</i>	<i>Resulting value</i>
bb12.34	F(7)	+12.34
+12.34b	F(7,1)	+12.34
12.34bb	F(7,3)	+12.34
bb1234b	F(7)	+1234
1234bbb	F(7,3)	+1.234
bbb1234	F(7,6)	+0.001234

The scaling factor  $p$  is used to bring the input number to the desired scale. If it is specified, the number is multiplied by  $10^p$ . Of the three format parameters  $w$ ,  $d$ ,  $p$ , only the scaling factor may be negative.

**Example**

<i>String of characters in input stream</i>	<i>Format specification</i>	<i>Resulting value</i>
b-23.7	F(6)	-23.7
b-23.7	F(6,1,1)	-237
b-23.7	F(6,0,1)	-237
b-23.7	F(6,0,-3)	-0.0237
bbb145	F(6)	+145
bbb145	F(6,0,-1)	+14.5
bbb145	F(6,3,1)	+1.45

At the output reversed conversions are made from internal to external representation. The result is rounded to the required number of digits after the decimal point and shifted to the right end of the field  $w$  characters wide. The plus sign and insignificant zeros in the integer part are suppressed by blanks. A minus sign is placed before the first significant digit or before the decimal point, if the integer part of the number is equal to zero. If the scaling factor  $p$  is specified, then before the conversion the number is divided by  $10^p$ .

If external representation of a number needs more characters than specified in the format, it is truncated on the left to  $w$  characters.

**Example**

<i>Value</i>	<i>Specified format</i>	<i>String of characters in output stream</i>
+3141.59	F(8,2)	b3141.59
-3141.59	F(8,2)	-3141.59
+3141.59	F(10,3)	bb3141.590
-3141.59	F(6)	b - 3142
+3141.59	F(4)	3142
-3141.59	F(4)	3142
+3141.59	F(4,1)	41.6
-3141.59	F(10,3,2)	bbb - 31.416

*E-floating-point format.* If the range within which a variable changes is unknown when the program is being written, then the floating-point format must be used either for computations, or for output. It is seen from the previous example, that in case of an improper choice of the F-format, the printed value may not correspond to the actual value of the variable.

Note that at the entry the E-format is used very seldom, mainly when we encounter numbers of the type 0.00000000145, which are more simple to be written in a floating-point form: 1.45E - 9.

The general form of the E-format element is as follows:

$$E(w, d [, s])$$

The specification of the format states that the current  $w$  characters of the stream are interpreted as a floating-point decimal number whose mantissa contains  $d$  digits after the decimal point. Like in the F-format, the specification  $d$  is suppressed if the point is explicitly punched in the number being read. The specification  $s$  standing for the total number of digits in the mantissa is ignored on input of information and may not be specified.

Numbers in the input stream must be punched according to the rules for writing PL/I floating-point constants in the form

$$[sign] \text{ mantissa } E [exponent-sign] \text{ exponent}$$

The mantissa is a fixed-point number, the exponent is a one- or two-digit decimal integer.

If on input the number record occupies less than  $w$  characters, then it may be located anywhere within the field  $w$  wide, adding the required number of blanks on the left and/or right.

### Example

<i>String of characters in input stream</i>	<i>Format specification</i>	<i>Value</i>
<i>b</i> +12.3E+02 <i>bbb</i>	E(13.1)	+1230
<i>bbb</i> 12.30E+2 <i>b</i>	E(13.7)	+1230
-123E0	E(6,0)	-123
-123E0	E(6,3)	-0.123
-123E1	E(6,3)	-1.23

On output, the number exponent is selected so that the high-order digit of the mantissa is not equal to zero. The mantissa is rounded to  $s$  digits of which  $d$  digits are to the left of the decimal point. The exponent of a number being output always consists of two digits, and the number itself is shifted to the right end of field  $w$  wide. The general form of a number output in the E-format is as follows:

$$\{\text{blanks}\} \quad [\text{sign}] \quad \{s-d \text{ digits}\}. \{d \text{ digits}\} \quad E \pm \{\text{exponent}\}$$

$$w - s - 6 + 1 \quad + s - d + 1 \quad + d \quad + 1 + 1 + 2 = w$$

If the total number of digits in the mantissa,  $s$ , is not specified, the number of significant digits to the left of the decimal point will be default equal to 1. The positive number sign is suppressed by a blank, and the exponent sign is always printed.

It follows from the form of floating-point number external representation, that in choosing the print format the following relationship must be observed

$$0 \leq d \leq s \leq w - 6$$

### Example

<i>Value</i>	<i>Format specification</i>	<i>String of characters in output stream</i>
+2.71828	E(13,6)	b2.718280E+00
-2.71828	E(13,6)	-2.718280E+00
+2.71828	E(12,4)	bb2.7183E+00
-2.71828	E(12,5,6)	-2.71828E+00
+2.71828	E(12,3,5)	bb27.183E-01
-2.71828	E(12,1,4)	bb-271.8E-02

**P ('picture specification')-format.** This format item is used seldom, usually when the F- or E-format print does not satisfy the user. In contrast to the previous formats, the P-format may be used to strictly describe each position of the output field. On input, the P-format is almost not used, as it imposes too severe restrictions on the rules for data punching. The P-format is commonly used in business computations on computers when data in a printed document is to be edited in some way.

The picture character string in the P-format is exactly the same as in the PICTURE attribute in the declaration of data in a statement.

### Example

<i>Value</i>	<i>Format specification</i>	<i>String of characters in output stream</i>
+123.45	P'99999'	00123
-123.45	P'999V99'	12345
+123.45	P'999V.99'	123.45
-123.45	P'S999.99'	-001.23
+123.45	P'S99999V.99'	+00123.45
-123.45	P'SZZZZZV.99'	-bb123.45
-123.45	P'-----V.99'	-123.45
+123.45	P'*****'	***123

**Character A-format.** This item of the format is used to describe fields in an input or output stream containing a string of characters. Generally, the A-element in the format list corresponds to a character variable or constant in the data list.

On input, the A-format is written in the form  $A(w)$ . The current  $w$  characters of the input stream are assigned to the corresponding variable in the data list. The assignment is in accordance with the rules for the execution of the assignment statement: if the variable's declared length attribute is greater than  $w$  characters, blanks are padded on the right; if the length is less than  $w$  characters, input data will be truncated on the right.

On output, if the format item is specified in the form  $A(w)$ , the reverse conversion occurs. If the format item is specified in A form, without specifying the field width, the output stream contains exactly as many characters as is the size of the corresponding variable or constant in the data list.

On output, the character string is not enclosed in apostrophes, like a constant in PL/I. On input, the apostrophes are treated as characters.

### Example

<i>Value</i>	<i>Format specification</i>	<i>Character string in output stream</i>
'ABCD'	A(4)	ABCD
'ABCD'	A	ABCD
'ABCD'	A(2)	AB
'ABCD'	A(6)	ABCDbb

*Bit B-format.* This format term is similar to the A-format, but is intended for handling bit strings.

On input, the format term should have the form  $B(w)$ . The current  $w$  characters of the input stream must contain a string of zeros and unities forming a bit string. If the length of the string is less than  $w$ , it may be padded with the required number of blanks on the left or right. No apostrophes and the letter B utilized in writing PL/I bit constants are required here.

### Example

<i>Character string in input stream</i>	<i>Format specification</i>	<i>Value</i>
101001	B(6)	'101001'B
b0110bb	B(7)	'0110'B
1bbbbbb	B(6)	'1'B
bbbbbb1	B(6)	'1'B

On output, a bit string is converted to a character string of 0's or 1's. If the format is specified in the form  $B(w)$ ,  $w$  characters are placed in the output stream. If the datum is less than  $w$  bits, it is padded with the required number of blanks on the right. If the datum size exceeds  $w$  bits, only left  $w$  bits are output.

If the format is specified in the B form, the number of output bits will be equal to the datum size.

### Example

<i>Internal value</i>	<i>Format specification</i>	<i>Output result</i>
'1001'B	B(4)	1001
'1001'B	B	1001
'1001'B	B(3)	100
'1001'B	B(7)	1001bbb

**Control items.** Recall that the control format items PAGE, LINE, or SKIP are allowed to be specified in the format of the PUT EDIT statement in order to position the output stream to the  $n$ th byte in the record or the  $n$ th print position on the line printer. These and certain other items may be specified in the format list to control the type of printout. A distinction between these two methods is that only one control item may be specified in the PUT statement, whereas a format list may contain as many control items as required.

No data in the data list correspond to the control format items.

**PAGE.** The PAGE format item causes a slip to the first print line of the next page.

**LINE ( $w$ ).** This format item specifies that the current data item is to be continued on the  $w$ th line. If the  $w$ th line has already been printed on a given page, a skip to a new page occurs.

**SKIP [ $w$ ].** This format item specifies the necessity to skip  $w - 1$  blank lines. If  $w$  is not specified, SKIP (1) is assumed by default, i.e. the printing is continued on the first position of the next line.

**COLUMN ( $w$ ).** On appearance of this item in the format list, the printing is continued on the  $w$ th position of the current line. If the  $w$ th position on the current line has been already printed, a new line is started and then a skip is made to the  $w$ th position. The skipped positions are padded with blanks in the output stream.

**X ( $w$ ).** On output, this format item causes  $w$  blanks to be inserted into the output stream.

Note, that the LINE and COLUMN items specify an absolute position in the list, and the SKIP and X items specify a position with regard to the current position of the print sheet.

The items of the COLUMN ( $w$ ), X ( $w$ ) format may be also specified in the GET EDIT statement. On appearance of the COLUMN ( $w$ ) item, the next in turn datum is read starting with the  $w$ th column of the current card, and if the  $w$ th column has already been read, then starting with the  $w$ th column of the next card. The skipped positions are ignored. If the X ( $w$ ) format is used,  $w$  positions in the input stream are ignored, regardless of the boundaries of the punched cards.

**Example.** The following information is punched:

<i>Column</i>	
1-20	Author of the book (the first name)
25-28	Year of edition
30-55	Book title
60-70	Book price in copecks

This information can be read by the following statements

```

DECLARE AUTHOR      CHARACTER (20),
        YEAR        FIXED DECIMAL (4),
        NAME        CHARACTER (26),
        PRICE       FIXED (5,2);
GET EDIT (AUTHOR, YEAR, NAME, PRICE)
(A(20), COLUMN (25), F(4), COLUMN (30), A(26),
COLUMN (60), F(11,2));

```

In this example blank positions between separate columns are skipped with the aid of the COLUMN format element. The same result may be obtained through the use of the X format:

```

GET EDIT (AUTHOR, YEAR, NAME, PRICE)
(A(20), X(4), F(4), X(1), A(26), X(4), F(11,2));

```

But the best result is obtained if the blanks between the columns are referred to one of the columns:

```

GET EDIT (AUTHOR, YEAR, NAME, PRICE)
(A(20), F(9), A(30), F(11,2));

```

The reader is recommended to return to the above descriptions of the format items to make sure that the last statement performs the same functions as the two previous statements. Note that the shorter the format list, the more efficient will be the program created by the compiler.

Let it be necessary to print the heading of the next in turn page of a document: the page No. in the right upper corner and the text INVENTORY SHEET in the middle of the third line. The page number is stored in the N variable. The solution is by the statement:

```

PUT EDIT ('SHEET', N, 'INVENTORY SHEET')
(PAGE, COLUMN (110), A, F(5), LINE(3), COLUMN (50), A);

```

Make notice of the use of constants in the data list. In this example SKIP (2) may be written in place of the format LINE (3), X (110) in place of COLUMN (110), and so on. Selection of a particular record is dependent solely upon the programmer's inclination.

**Repetition factor in the format list.** Suppose that a program contains the declaration:

```

DECLARE A(10) CHARACTER (1),
        B(10) CHARACTER (4);

```

The values of the elements of arrays A and B are punched in the following sequence:

A (1), A (2), . . . , A (10), B (1), . . . , B (10)

It is necessary to read in the values of the elements of array A and B.

The data list for the input statement is easy to write. It will do to specify only the array identifiers: (A, B). However, in fact, the list will comprise twenty simple data, for which reason the format list must also include twenty items.

Naturally, it is not advisable to write up all the twenty formats, as there are many similar formats among them. Therefore, PL/I allows the number of occurrences of similar formats to be written by means of the repetition factor in a short form.

With our example, the input statement may take the form:

GET EDIT (A, B) (10 A (1), 10 A (4));

The record 10 A (1) is equivalent to ten A (1) format items written in succession.

The repetition factor applies both to format items and to occurrences of format item groups. Let the data under the conditions of our example be punched in the following sequence:

A (1), B (1), A (2), B (2), . . . , A (10), B (10)

Then to read them in for printing, use may be made of the statement

GET EDIT ((A (N) B (N) DO N = 1 TO 10)) (10 (A (1), A (4)));

If a group of format items is to be repeated, it must be parenthesized, and the repetition factor must be specified preceding the left parenthesis.

**Interaction of the data list and format list.** In the previous examples, the number of data in the lists of input/output statements must exactly match the number of format items (less the control format items). The programmer should remember the rule stating that this requirement must always be met. However, PL/I permits data lists and format lists of different lengths, which may be useful in certain tasks.

If that is the case, the following syntax rules must be followed:

1. If there are more format items than data items, the extra format items are ignored. Thus, on the statement

PUT EDIT (S, T) (SKIP), F (8, 5), X (2), A (10), PAGE, A (30));

a skip to a new line will occur. The value of variable S will be printed with format F (8, 5), then two blanks will be inserted, and the value of variable T will be printed with format A (10). This will ter-

minate the operation and no skip to a new page will be accomplished on the PAGE control option.

2. If there are less format items than data items, there is a return to the beginning of the format list. This feature provides programming of large data lists with a short format list.

Let a one-dimensional array PRIME whose elements are whole numbers not in excess of 9999 in absolute value be specified in a program. Print the values of the first N elements of the array arranging them ten in each row. The solution is by the statement

```
PUT EDIT ((PRIME (I) DO I = 1 TON)) (SKIP, 10F (6));
```

Consider in more detail the actions involved in this operation. On the control item of the SKIP format, a skip will occur to a new line. Then, the first ten elements of the PRIME array will be printed. At this point, the format list will be completed, while the data list will still remain unexhausted. Therefore, the format list will be scanned again, a skip will occur to a new line on the SKIP term, next ten elements of the array will be printed, and so on. If the number N is not divisible by 10, then in printing the last line, the data list will be exhausted earlier than the format list, and the remaining format items will be ignored.

**Punched card files.** In the edit-directed input/output, the punched cards are considered to be in the form of a continuous stream of characters, the boundaries separating one card from another being not marked at all. However, in practice punching information in this way is inconvenient. Punched cards as an information-carrying medium are advantageous in that the files are easy to be updated. It is sufficiently simple to remove an erroneous card or replace it with a correct one, insert missed cards where it is required, etc. Therefore, each independent block of information is usually punched into a separate card at the end of which unused columns may be left. This circumstance must be taken into account in building programs which utilize a card input.

**Example.** A goods price list is prepared on punched cards in the following format:

*Columns*

1-9	Goods index
10-44	Goods name
45-52	Price in copecks

Information about each kind of goods is punched into a separate card. Write statements to read the card file into the storage. The natural decision is as follows:

```

DECLARE                                INDEX FIXED (9),
                                         NAME CHARACTER(35),
                                         PRICE FIXED(8);
NEXT:      GET EDIT (INDEX, NAME, PRICE)
                                         (F(9), A(35), F(8));
                                         . . . . .
GOTO NEXT

```

This decision is erroneous, as on the first statement GET, the machine will scan only 52 columns of the first card. The next statement GET will attempt to read information on the second kind of goods from column 53 of the first card, rather than from the first column of the second card.

This error has not been avoided by most of the programmers starting their PL/I work. This is especially true of those who programmed before in FORTRAN, since in FORTRAN each statement GET automatically starts its operation from a new card.

A hasty attempt to correct the error by specifying the GET statement in the form

```
GET EDIT (INDEX, NAME, PRICE) (F(9), A(35), F(8), X(28));
```

will not be successful, for there are more format items than data items, and the item X (28), intended for passing the last 28 columns of the card will not be used.

There are several methods of passing the last columns of a card, but we shall consider one, most effective. Insert a dummy character variable one character long:

```
DECLARE DUMMY CHARACTER(1);
```

and write the input (GET) statement in the following form:

```
GET EDIT (INDEX, NAME, PRICE, DUMMY) (F(9), A(35), F(8), A(28));
```

Note that the sum of fields processed according to the format items is equal to 80 columns, i.e. one card. It is good practice to have format lists in the GET EDIT formed only in the manner illustrated. This will guarantee you against reading information from other than specified columns of a card.

To conclude this section, let us mention as an example a complete program which will well illustrate the wide abilities of PL/I which allows the desired results to be obtained at minimum costs.

It is necessary to print a table of squared whole numbers from 0 to 9999 in as compact form as practicable. Print the numbers in 10 columns 50 lines each per page, i.e. 500 numbers per page. Let printed in the left column be the quantity of tens in the number being squared, and above the column, the quantity of unities. To make reading easy, skip a blank line after each 10 lines. Assign numbers to

Table 8.1

				Sheet 1	
	0	1	2		9
0	0	1	4		81
10	100	121	144		361
				...	...
90	8100	8281	8464	...	9801
100	10000	10201	10404	...	11881
				...	...
490	240100	241081	242064	...	249001

the table sheets (there will be 20 sheets) by printing the sheet number in the right upper corner. For the format of a table sheet, see Table 8.1.

The way this problem is stated is very simple, but the resultant program seems fairly awkward. Indeed, a loop is necessary to turn over a page. A heading must be printed on each page, while printing a page itself is a double loop dealing with the table lines and columns. More than that, the work is hindered by the requirement to skip a blank line after each ten lines. However, it is surprising that in reality all the above required actions can be accomplished by one output statement, though fairly large:

```
SQTAB: PROCEDURE OPTIONS(MAIN);
  DECLARE (K, L, M, N) BINARY FIXED (31);
  PUT EDIT (('SHEET', K, (L DO L=0 TO 9), (M,
(N*N DO N=M TO M+9) DO M=500*(K-1)
BY 10 TO 500*(K-10) DO K=1 TO 20))
(PAGE, X(110),A,F(3),SKIP(2),X(10),10F(10),
SKIP,5 (SKIP,10(SKIP,11 F(10))));
END SQTAB;
```

This program utilizes almost all the programming tools described in the section. In examining the program, pay special attention to the interaction of the data list considering its built-in loops and the format list with the repetition factors taken into account.

## CHAPTER 9

### INTRODUCTION TO FORTRAN

#### 9.1. Special Features of FORTRAN

FORTRAN (FORmula TRANslator) is one of the oldest and most widely used of the computer algorithmic languages for programming engineering and scientific problems.

A specific feature of FORTRAN is that the structure of this algorithmic language allows you to obtain good-quality operating programs after assembling. This language is convenient for program segmentation with off-line translation of segments, and, besides, it permits the use of different data formats in input/output referencing. The FORTRAN procedures practically cover all generally used methods of computations and processing.

FORTRAN was developed in 1954. The language was oriented toward the field of scientific and engineering computations. FORTRAN is very close to the language of regular algebra, while its vocabulary originates from the English.

Problems of solving game situations, simulating intellectual activity of people and production processes, plotting curves from computation results, etc. are easy to be described in FORTRAN.

The merit of FORTRAN is that it allows persons who have no chance to study the machine language in detail to solve problems on a computer. The wide spreading of FORTRAN has lead to its use on many types of computers, hence FORTRAN compilers are available on many computers. However, it must be borne in mind that FORTRAN may not be used on whatever machines without taking their specific features into consideration. In some aspects it is dependent upon the design of the machine.

The FORTRAN compilers make use of certain specific features of the computer and are dependent upon the computer type. In such cases as these, additions should be made to the basic language. In designing compilers, special attention should be paid to the efficiency of translation (its quality and speed). The quality of translation (compilation) is dependent upon the detection of syntax and logic errors, apt solution of information input/output problems, and a compiler structure well suitable for debugging purposes.

## 9.2. Basic Elements of the Language

The words and expressions of FORTRAN are based on three elements: letters, digits and special characters.

The special characters of FORTRAN are

$$+ \ - \ * \ / \ = \ ( \ ) \ , \ .$$

There is a separate *blank* symbol which either has no designation notation, or is represented by the letter *b*.

The first four characters of the above-listed special characters stand for arithmetic operations.

The equals sign  $=$  in FORTRAN is not used to indicate mathematical equality but is strictly an *assignment symbol*. It tells the compu-

ter to assign the value of the expression on the right of the symbol to the variable on the left of the symbol.

Thus,  $A = B$  in FORTRAN tells the computer to assign the value of  $B$  to  $A$ .

Parentheses are used in FORTRAN to group arguments and subscripts, or to specify the sequence of executing arithmetic and logic operations.

A comma is a separator in lists of arguments or subscripts.

A period is used as a separator of elements in logic expressions and a decimal point.

An asterisk is used as a multiplication operator (\*) and exponentiation operator (\*\*).

Words in FORTRAN are formations of letters and digits of the character set. The words are divided into *keywords* and *names*.

The keywords have a strictly specified spelling and meaning. The programmer may use them only in the meaning pre-defined in the FORTRAN grammar. Blanks are permitted in keywords. Keywords are elements of the FORTRAN alphabet.

The basic keywords are listed below:

CALL	GO TO
COMMON	IF
COMPLEX	INTEGER
CONTINUE	LOGICAL
DIMENSION	PAUSE
DO	READ
DOUBLE PRECISION	REAL
END	RETURN
EQUIVALENCE	STOP
FORMAT	SUBROUTINE
FUNCTION	WRITE

This list does not cover all keywords in use.

The keywords may be subdivided into the following groups:

—The words of imperative nature to carry out some action: DO, CALL, READ;

—The words which describe quantities: INTEGER, LOGICAL, REAL;

—The words which define the language structure divisions: SUBROUTINE, FUNCTION;

—The words that describe groups of values and their relationship: COMMON, DIMENSION.

In addition to the above-listed keywords, FORTRAN includes words which are relational operators and logical operators, and words naming standard library programs. Table 9.1 covers the words which are relational operators.

Table 9.1

Relational operator	Meaning	Mathematical symbol
.GT.	greater than	$>$
.GE.	greater than or equal to	$\geq$
.LT.	less than	$<$
.LE.	less than or equal to	$\leq$
.EQ.	equal to	$=$
.NE.	not equal to	$\neq$

### 9.3. Numeric Constants

We shall deal with two types of constants. These are integer constants and real constants. The integer constants are literally integers, i.e. whole numbers without a decimal point. An integer constant may be preceded by a sign, and is written as a string of digits. No decimal point, comma or exponent may appear. The plus sign (+) is not commonly present and the constant is assumed to be positive.

Examples of writing integer constants (numbers) are  $-26$ ;  $0$ ;  $3\ 964\ 728\ 654$ .

The real, or floating-point, constants are used to write rational numbers. For the forms in which real constants (numbers) can be expressed, see Table 9.2, where  $k$ ,  $m$ ,  $n$  are integer nonnegative numbers.  $E \pm k$  means multiplying by  $10^k$ .

Table 9.2

General form of writing	Example of FORTRAN notation	Usual notation
$n \cdot m$	$2 \cdot 35$	$2 \cdot 35$
$n \cdot$	$42312 \cdot$	$42312 \cdot$
$\cdot m$	$\cdot 241$	$0 \cdot 241$
$n \cdot m E \pm k$	$61 \cdot 197 E 5$	$61 \cdot 197 \cdot 10^5$
$n \cdot E \pm k$	$247 \cdot E - 7$	$247 \cdot 10^{-7}$
$\cdot m E \pm k$	$\cdot 02394 E 12$	$0 \cdot 2394 \cdot 10^{11}$

### Exercises

1. Following the FORTRAN rule, write the following numbers in the form of real constants:

$-10^{-15}$ ;  $124$ ;  $-72.64$ ;  $-0.7$ ;  $0.000000724$ ;  $32 \cdot 10^6$

2. Decode the following numbers written in the form of real constants:

379.0017; -694; 0039; 2.6E - 11; -2998.E7; 0.000087E - 14

3. Say whether the following pairs of constants are the same values:

+339.7	and	339.7;
16400	and	164.E2;
0.17	and	1.7E1;
0.93716	and	.93716;
2.0	and	2.;
.141E7	and	+141.E+4

4. Explain why the following designations are not acceptable integer constants:

-19.; 37.E3; 193,0; 75E - 3; -0.001

### 9.4. Simple Variables

A *variable* is a single item of data whose value may be changed in the course of a program.

In the FORTRAN algorithmic language each variable is referred to by a name. A name may consist of uppercase Roman letters, and also any combinations of letters and digits in which the first character must be a letter and which must not exceed six characters.

For example:

X, Y64, ZIS 101, A110, BETA, GAMMA, NN17, LUNA, R25MAM

The variable will be of

INTEGER type — if its name begins with the letter I, J, K, L, M or N

or REAL type — if its name begins with any other letter of the alphabet

#### Exercises

1. Explain what of the following names are acceptable integer and real variables: ALFA, DUBNA1, KAPPA, NIL, FINT, PTO23A, I, ROZA, LASZ, BOBBY.

2. Explain why the following records cannot be names of variables: XY + Z, 12BEE, A \* B, M1.7, FORTRAN.

### 9.5. Arithmetic Operations and Expressions

FORTRAN provides for five basic arithmetic operations on numeric constants, variable values, and functions each of which is represented by an easily distinguishable symbol:

- + addition
- subtraction
- \* multiplication
- / division
- \*\* exponentiation

Arithmetic expressions are:

(a) Numeric constants. Examples are 25;  $-.126$ ;  $2.5E-4$

(b) Variable values, for example:

X, NOP, U18MK

(c) Functions, for example:

SIN (X), EXP (Z), RIGHT (X, Y), and

(d) Numeric constants, variables and functions combined by using arithmetic symbols of operation and/or parentheses, for example:

$A + (B21 * X - 25) ** 2 - C5/4.6$

In writing FORTRAN arithmetic expressions the following rules must be observed:

1. Within a set of parentheses or in the expression as a whole, the order of evaluation is as follows: first, all operations of exponentiation, then all operations of multiplication and division, and finally all operations of addition and subtraction. If several multiply and divide operators are encountered in an expression in succession, the operations are performed from left to right also in succession.

The same rule applies to addition and subtraction operations.

For example, an expression  $A/B * C$  means  $\frac{AC}{B}$ , rather than  $\frac{A}{BC}$ , while an expression  $X - Y + Z$  means  $(X - Y) + Z$ , but not  $X - (Y + Z)$ .

2. Parenthesized expressions are evaluated first of all. If an expression containing elements consisting of other arithmetic expressions enclosed in parentheses is parenthesized itself, then evaluation is made from inside out. For example, in an expression  $A * (X - B / (X + C))$ , find first the sum  $X + C$ , then the quotient  $Y = \frac{B}{X+C}$ , then the difference  $Z = X - Y$ , and finally the product  $A * Z$ .

3. Operators must not be adjacent to each other (\*\* appears to be an exception to this rule, but is a single operator): they must be separated by variables, constants or parentheses in an expression. Neither the multiplication sign may be omitted between the multipliers. For example, an algebraic expression  $4AB$  and  $\frac{X}{-Z}$  in FORTRAN should be written as follows:  $4 * A * B$  and  $X / (-Z)$ ; records  $4AB$  and  $X / -Z$  are invalid.

4. In evaluation of expressions consisting of integer values, it should be remembered that arithmetic operations are performed

with dropping fractional parts in the intermediate results. For example, evaluating an expression  $4 * (7/2)$  will give us 12, since the quotient resulting from dividing 7 by 2 equals 3, rather than 3.5 as the case is with ordinary arithmetic.

### Exercises

1. Write arithmetic expressions corresponding to the following formulas:

$$(a) (A + B)^4; \quad (b) A + \frac{B}{C}; \quad (c) \frac{X+2}{Y+4}; \quad (d) \frac{A+B}{C + \frac{D}{F+G}};$$

$$(e) \left\{ [A + 2B]^2 - \frac{A}{4} \right\}^8 + 3B^2 \Big\}^3;$$

$$(f) 2.6 \cdot 0.134 \cdot 10^{-6} + \left( 4AX - \frac{34.6 \cdot 10^{12}}{732A} \right)^{-1}$$

2. Write mathematical formulas corresponding to the given arithmetic expressions:

$$(a) M ** (K + 3); \quad (b) A + B / (C * D);$$

$$(c) X * Y / C * D; \quad (d) ((A + B) / C) ** 3.2;$$

$$(e) (A + 2) * (X + B * (Z + C - 3.7))$$

3. Find errors in the following arithmetic expressions which should correspond to the mathematical formulae found on the right:

$$(a) K + L ** 4 \qquad (K + L)^4$$

$$(b) X + 7./Y - 6 \qquad \frac{X+7}{Y-6}$$

$$(c) (A + B + C) / (2 * Z) ** 3 \qquad \left( \frac{A+B+C}{2Z} \right)^3$$

$$(d) (X/Y) ** (R - 2) \qquad \left( \frac{X}{Y} \right)^{R-2}$$

## 9.6. Elementary Mathematical Functions

To calculate certain elementary functions such as sine, cosine, exponent, square root, absolute value and others, we should write the name of the corresponding function and its argument enclosed in parentheses. Some functions are tabulated below:

<i>FORTRAN record</i>	<i>Mathematical function</i>
SIN(X)	SIN X, X in radians
COS(X)	COS X, X in radians
EXP(X)	$e^x$
SQRT(X)	$\sqrt{X}$ , $X \geq 0$
ABS(X)	$ X $

Note that the argument  $X$  may be any arithmetic expression, whose numeric value is permissible for a given function.

**Example.** In order to find the absolute value of the expression  $(A^2 - 13.7 BC)^3$ , it is enough to write

ABS ((A \*\* 2 - 13.7 \* B \* C) \*\* 3)

### 9.7. Arithmetic Assignment

Let  $A$  be the name of variable and  $E$  be an arithmetic expression, then  $A = E$  is a general form of the arithmetic assignment statement.

The statement  $=$  tells the computer to evaluate the expression  $E$  and to assign that value to the variable  $A$ . For example, the statement  $A = A + B$  means that first the expression  $A + B$  is evaluated and then the previous value of the variable  $A$  is substituted for by the sum obtained.

The arithmetic assignment statement may also be used to change integer values to real values and vice versa. Thus, if an expression consisting of integer constants and integer variables is written to the right of the symbol  $=$ , and the name of a real variable is to the left of the symbol, then the evaluations will be made in compliance with the rules for operations on whole numbers, and the result will be converted into a real form and assigned to the variable found in the left term.

#### Exercises

1. Determine the values of the variables  $A$  and  $K$  resulting from the execution of the following arithmetic assignment statements:

- (a)  $A = 2/7$ ; (b)  $K = 19/4 + 5/4$ ; (c)  $A = 19/4 + 5/4$ ;  
 (d)  $K = 19.0/4.0 + 5.0/4.0$ ; (e)  $A = 19./4. + 5./4.$

2. Find errors in the following records of arithmetic statements:

- (a)  $A = F - 2.Y$ ; (b)  $-K = (X - Y) * - * 3$ ; (c)  $3.175 = Z + B$ ;  
 (d)  $A * B - 2 * C = R$ ; (e)  $\text{COS}(X) = 1. - X ** 2/2.$

### 9.8. Writing a Program on Coding Sheet and Punching It into Cards

Each program written in the FORTRAN language consists of statements some of which have labels. A label is a whole number used to identify a statement which is to be referenced elsewhere in the program. A program is generally written on a coding sheet (form) each line of which contains 80 positions.

When writing a program on a coding sheet, the following rules should be observed.

1. Not more than one character (a letter, digit, period, comma, parenthesis, and the like) must be written in each position (column) of the coding form. Certain positions may be left unfilled.

2. Each statement is normally written on a single line only in columns 7 through 72.

3. Where statements are too long to be contained on a single line, they may be continued to the next line. This continuation is indicated in column 6. Any character except a blank or zero in this column denotes that the statement on the preceding line is continued on this line.

4. Columns 1 through 5 are used only for statement labels (numbers).

5. Columns 73 through 80 may optionally be used for program identification or sequence numbers. The contents of these columns are ignored by the FORTRAN compiler.

To load a program in a computer, use is made of punched cards. With the aid of a special device (a card punch unit) each line of a program written on a coding sheet is punched into a single card.

### 9.9. Input of Numeric Data

One of the most important applications of most programs is mathematical processing of specified numerical values (experimental data, for instance) and obtaining of results in a numerical form. The execution of a program involves execution of its statements in sequence. Prior to execution of each statement, all its variables (except those to be computed by a given statement) must be assigned actual numerical values. This can be accomplished, say, with the aid of the arithmetic assignment statements.

The input of numerical data with the help of arithmetic assignment statements is commonly used when these numerical data are constant for the program in question, or when the program is used only once with the small quantity of numerical data.

In cases when many numbers have to be used, or when the program is used many times, numerical data are commonly read in from cards with the aid of a special input statement, which has the following form:

READ (*u*, *f*) *l*

where *u* is a conventional number of the card reader; *l* is the list of the variables the new values of which must be read from the cards; *f* is the label (number) of the FORMAT statement which works together with the READ statement and describes how the numerical data have been punched into the cards.

A comma is used to separate the elements of list *l* from each other. The new values to be assigned to the variables must be punched in-

to the cards in the same sequence as their names are written in list  $l$ . During the input of numerical data, use may be made of all 80 columns of a card.

The specification of the field for each variable from list  $l$  must be specified in the FORMAT statement within parentheses.

The field specification for integer constants has the form  $I_n$  where  $n$  stands for the field width, i.e. the number of card columns allocated on the card for punching the corresponding integer constant.

The field specification for real constants is in the form of  $F_{n,m}$  or  $E_{n,m}$ , the former of which is utilized if the real constant is in the form of  $n.m$ ,  $n.$  or  $.m$ , and the latter one is utilized if the real constant is in the form of  $n.mE \pm k$ ,  $n.E \pm k$  or  $.mE \pm k$ . If the field specification has the form of  $E_{n,m}$  the numbers  $n$  and  $m$  must be so selected that  $7 + m \leq n$ .

Commas are used in the FORMAT statement to separate the specifications from each other. If a specification is to be repeated several times, a number standing for the number of repetitions written to precede the specification will do. A group of specifications may be also repeated. To this end that group must be parenthesized, with the repetition number to precede the left parenthesis specified. If during the input of the numerical data, all specifications of the FORMAT statement turn out to be utilized, though not all the variables of the list have been assigned numerical values, then, in the absence of inner parentheses in the FORMAT statement, its specifications are reused starting with the first specification. If the numerical data for list  $l$  are punched into more than one card, the special character / is used in the FORMAT statement to specify reading of a new card. A skip to reading a new punched card (list  $l$  still contains elements to be assigned new values) occurs when the use of the FORMAT specification reaches the right parenthesis.

**Example.** Input numerical values from cards for variables X, A, B, N, Y, C, D, M, if it is known that the values for the X, A, B are punched into the first card, for the N, into the second card, for the Y, C, D, into the third card, and for the M, into the fourth card. Besides, we know that:

(a) the variables X and Y occupy the first six columns of the corresponding cards, using an F specification, with two digits after the decimal point;

(b) the variables A, B, C, and D occupy eight subsequent columns each, with one digit after the decimal point;

(c) the four first columns of separate cards are used for integer variables N and M.

**Solution:** The problem in question can be solved, say, with the aid of the following statements:

```
READ (3,5) X, A, B, N, Y, C, D, M
FORMAT (F6.2, 2F8.1/14)
```

**Exercises**

1. Explain the purpose and work of the following pairs of statements:

- (a)    READ (3,1) A, B, C  
      1    FORMAT (3E10.2)
- (b)    READ(3,2) A, B, C  
      2    FORMAT (F14.5)
- (c)    READ(3,3) X, Y, B1, PP, K  
      3    FORMAT (2F8.3, E12.4/F10.4/15
- (d)    READ(3,4) AT, F1, M, X, Y  
      4    FORMAT(E12.3, F6.0/110)

2. Read numerical data from punched cards:

(a) For variables A, X, Y, Z the numerical values of which are punched into one card where each number is assigned a field of ten columns. The numbers are represented as real constants with no exponents, accurate to three places.

(b) Same as at (a), but the numerical values for each variable are punched into a separate card.

(c) Same as at (a), but the numerical data are represented using an exponent.

(d) For variables A, B, NOT, KAT, if it is known that the numerical values for each variable are punched into separate cards, 15 columns being assigned to each number. The real constants are accurate to three places and contain no exponents in their record.

**9.10. Information Printout**

The information printout can be accomplished with the aid of a write statement which takes the form

WRITE (*u*, *f*) *l*

where *u* is the conventional number of the printer; *l* is a list of variable names whose values are to be printed; and *f* is the label of the FORMAT statement used in conjunction with the WRITE statement to specify how the results are to be printed.

The printing speed is measured in lines per time. The number of characters per line is dependent on the type of the printer in use.

In printing out, the WRITE statement works together with the FORMAT statement in a way similar to the interaction between the READ statement and FORMAT statement in reading information from punched cards. To print out numerical constants use is made of the same specifications (I, F, E) as in case of input. A skip to a new line of printing occurs when a symbol / is encountered, and also when the use of the FORMAT specification reaches the right parenthesis. In forming a string with the aid of the FORMAT statement, it should

be kept in mind that the first position (column) of each line is not printed out. This is a paper control character: a unity in the first position advances the paper to the first line of the next page, a blank in it advances the paper one line (single space), and a zero in the first position advances the paper two lines (double space).

During information printout, use is often made of an  $nX$  specification which instructs the printer to make  $n$  blanks in a line, and also of an  $nH$  specification which tells the printer to use the subsequent  $n$  positions of a line for printing those characters (blanks included) which immediately follow the letter H in the FORMAT statement. Another method of printing a text is by writing it in the FORMAT statement and enclosing the record by quotes.

### Exercises

Explain the purpose and working of the following pairs of statements:

- (a) WRITE (2.5) A, B, X  
5 FORMAT (40X, 12.5)
- (b) WRITE (2.6) X, Y  
6 FORMAT (10X, 2HX = , F8.2, 10X, 2HY = , F8.2)

## 9.11. Sample Examples of Simple Programs in FORTRAN

The information data on FORTRAN set forth in the text previously are enough to write simple programs. A program must terminate with an END statement.

**Example.** Write a program to calculate the area of a triangle by the known lengths of its sides:  $A = 3$ ,  $B = 4$ ,  $C = 5$ .

**Solution.** Using the Heron's formula, the program may be written as follows:

```

A = 3.
B = 4.
C = 5.
P = (A + B + C)*.5
S = SQRT (P*(P - A)*(P - B)*(P - C))
WRITE (2,5) A, B, C, S
5  FORMAT (3HbA=, F6.2,5X, 2HB=, F6.2,5X,
+2HC=, F6.2,5X, 2HS=, F10.4)
END

```

The plus sign (+) in the FORMAT statement means that the statement is continued on the next line. The character  $b$  in the first specification of the format stands for a blank.

**Example.** Write a program for calculating the area of a triangle, if it is known that the numerical values of each of the triangle sides

contain no more than three decimal characters after the decimal point and are punched into separate cards using the first six columns of each card.

**Solution.** The program may be built as follows:

```

      READ(3,2) X, Y, Z
2   FORMAT(F6,3)
      P = (X + Y + Z)*5
      S = SQRT(P*(P - X)*(P - Y)*(P - Z))
      WRITE(2,3) X, Y, Z, S
3   FORMAT(3HbA=F8,3/3HbB=, F8.3
      + /3HbC=, F8,3/3HbS=, F8.3)
      END

```

**Exercise.** Write a program to solve the quadratic equation  $AX^2 + BX + C = 0$ , if it is known that  $B^2 - 4AC > 0$ .

### 9.12. GOTO Statement

If a GOTO  $n$  statement occurs in a program, where  $n$  is a natural number, then the statement labeled  $n$  is executed next after the GOTO statement.

### 9.13. Arithmetic IF Statement

The general form of the arithmetic IF statement is

IF (E)  $n_1, n_2, n_3$

where E is a certain arithmetic expression; and  $n_1, n_2, n_3$  are statement labels.

This statement computes the values of E and then passes control according to the following rule:

- If  $E < 0$ , the control is passed to the statement having label  $n_1$ ;
- If  $E = 0$ , the control is transferred to the statement labeled  $n_2$ ;
- If  $E > 0$ , the control is passed to the statement having label  $n_3$ .

**Example.** The program segment below assigns the variable C the greater of the values of the variables A and B

```

      IF(A-B)1, 2, 2
1   C=B
      GO TO 3
2   C=A
3   Continuation of the program

```

**Exercises**

1. Write a program to solve the quadratic equation

$$AX^2 + BX + C = 0 \text{ with } A \neq 0, B \neq 0, C \neq 0$$

2. Write a program for computation of the function table  $f(X) = 2X^2 - 3X + 5$  in 11 points, if  $X_1 = 0$ ,  $X_{k+1} = X_k + 0.1$  where  $k = 1, \dots, 10$ .

**9.14. Variable Arrays**

In addition to simple variables, used in FORTRAN often are subscripted variables. The subscripts allow a great number of quantities to be represented by one common name. Each subscripted variable is defined by one, two or three subscripts enclosed in parentheses which immediately follow the variable, for example:  $A(2)$ ,  $M(3,7)$ ,  $Z(2, 5, 4)$ .

The names for subscripted variables are selected following the same rules as for simple variables. A complete set of subscripted variables under a common name is called an *array*, and each subscripted variable is known as an *array element*. The array is designated in a way similar to its elements, but without subscript. For example, a subscripted variable  $A(2)$  corresponds to the second element of an array  $A$ . An array comprising variables with one subscript is called a one-dimensional array, with two subscripts, a two-dimensional array, and with three subscripts, a three-dimensional array. In a program, arrays must be declared in a DIMENSION statement which specifies their dimension.

The DIMENSION statement is a nonexecutable statement. In a program it is written before any executable statement in the following form:

DIMENSION A (...), B (...), C (...)

where  $A, B, C$  are the names of arrays followed by parentheses containing one, two or three real numbers. The quantity of these numbers define the array dimension, while their values define the maximum value of each subscript for those subscripted variables which are elements of the corresponding array.

For example, the statement

DIMENSION A (4), B (3,2)

means that the array  $A$  consists of four elements:  $A(1)$ ,  $A(2)$ ,  $A(3)$ , and  $A(4)$ , and the array  $B$  of six elements arranged in three rows, two elements in each:

$B(1, 1) \quad B(1,2)$

$B(2, 1) \quad B(2,2)$

$B(3, 1) \quad B(3,2)$

The elements of a two-dimensional array may also be specified by variables with one subscript. In this case, the subscript indicates the sequential number of the element in the array when the elements are counted in the columns. Thus, a variable with two subscripts B (2,2) can be designated by variable with one subscript B (5).

Not only natural numbers may be used as subscripts, but also variables of an 'integer' type which have been assigned positive values in the course of the program run, and also expressions of an integer type.

Note that subscripted variables can be included in list  $l$  of READ and WRITE in three ways.

1. Variables may be written with indication of their subscripts. For example, the statement

WRITE (2, 5) A (2), B (3, 4)

means that the numerical values of the variable A (2) of the array A and variable B (3,4) of the array B will be printed in compliance with the FORMAT statement which has label 5.

2. The list  $l$  may contain only the name of the array. If that is the case, the list includes all elements of a given array in the order in which they are arranged in the array (in case of a two-dimensional array, the counting is made by the columns). For example, if A and B are certain arrays declared in the statement

DIMENSION A (3), B (2, 3)

then the statement READ (3, 7) A, B instructs the computer to assign the numerical values from the punched cards to the variables of the arrays A and B according to the FORMAT statement in the following order:

A (1), A (2), A (3), B (1, 1), B (2, 1), B (1, 2), B (2, 2), B (1, 3),  
B (2, 3)

3. Variables may be included in list  $l$  with subscript values defined by means of an implicit loop.

With one-dimensional arrays, a record of such a loop takes the form

A (J), J =  $m_1, m_2, m_3$

where  $m_1, m_2, m_3$  are real numbers or integer variables which have been assigned the values of real numbers. This record means that list  $l$  does not include variables from the array A whose subscripts are equal to  $m_1, m_1 + m_3, m_1 + 2m_3$ , and so on, until the subscript remains less than, or equal to,  $m_2$ .

For example, the statement

WRITE (2, 8) (B (N), N = 1, 7, 2)

tells us that the printer will print (according to the FORMAT statement labeled 8) the values of the variables having subscripts B (1), B (3), B (5), B (7).

For a two-dimensional array, the record of an implicit loop may take the form

$$((A(I, J), I = m_1, m_2, m_3), J = n_1, n_2, n_3)$$

or

$$((A(I, J), J = n_1, n_2, n_3), I = m_1, m_2, m_3)$$

In the former case, variables are selected by columns, and in the latter case by the rows.

For example, the statement

$$\text{WRITE}(2, 100) ((X(M, N), N = 1, 3, 2), M = 2, 4, 2)$$

tells us that the printer will print, according to the FORMAT statement having label 100, numerical values of the following subscripted variables from the two-dimensional array X:

$$X(2, 1), X(2, 3), X(4, 1), X(4, 3)$$

### Exercises

1. Given: the first term of an arithmetic progression  $a_1$  and difference  $d$ . Using the formula  $a_k = a_{k-1} + d$ , find terms 2 through 10 of this progression and place them in a one-dimensional array A.

2. A one-dimensional array B consisting of 100 elements is filled with real numbers. Assign the value of the maximum number of the array A to the variable X.

3. Write a program for computing the values of the function  $f(X) = e^x + 3X - 1$  at 12 points  $X_i$ , if all  $X_i$  are punched into one card.

4. Print out line by line the numerical values of two-dimensional array A constituted by five rows, six elements in each.

### 9.15. DO Statement

A DO statement causes a group of statements immediately following it to be executed repeatedly. The general form of the DO statement is

$$\text{DO } n \text{ I} = m_1, m_2, m_3$$

where  $n$  is the label of the terminal (last) statement associated with the DO statement (this is commonly a CONTINUE statement); I is the control variable (an integer variable);  $m_1, m_2, m_3$  are natural numbers ( $m_1 \leq m_2$ ), or integer variables which must be assigned the values of certain natural numbers (must be greater than zero) at the time of execution of a DO statement.

The DO statement, the statement with the label  $n$ , and all the statements between them comprise a loop. The statements of the given loop are first executed at  $I = m_1$ ,  $I = m_1 + m_3$ , then at  $I = m_1 + 2m_3$ , and so on, until  $I \leq m_2$ .

If  $m_3 = 1$ , the DO statement may be written as follows:

DO  $n$  I =  $m_1$ ,  $m_2$

Note that within the range of a DO statement there may be another DO statement with its range. If during execution of the loop, control is passed to a statement outside the loop, then the current value of the control variable is saved. If the execution of the loop is completed, the value of the control variable  $I$  becomes undefined.

**Example.** Raise to square power each of  $K$  numerical values contained in the array  $A$ .

**Solution.**

```
DO 1 M = 1, K
  A (M) = A (M) ** 2
1  CONTINUE
```

*Note.* If the CONTINUE statement is not the terminal statement of the loop, it is ignored in the course of the program.

#### Exercises

1. A one-dimensional array  $I$  consists of 200 elements. Square each element of the array found in an odd position, and raise each element in an even position to the fifth power.

2. Find the sum  $C$  of square matrices  $A$  and  $B$  each of which has  $n$  rows and  $m$  columns.

3. Find the product  $C$  of square matrices  $X$  and  $Y$ , exponent  $n$ .

### 9.16. FUNCTION Subprogram

The FUNCTION subprogram is used when a certain function  $F(X_1, \dots, X_n)$  is to be evaluated many times at the corresponding values of the variables  $X_1, \dots, X_n$ .

The FUNCTION subprogram is formatted as follows. First, write a statement

FUNCTION  $F(X_1, X_2, \dots, X_n)$

where  $X_i$  ( $i = 1, 2, \dots, n$ ) are dummy arguments, i.e. those arguments of the function  $F$  which are not assigned actual numerical values, and  $F$  is a function name. It is chosen in the same manner as the names of simple variables. Next, a group of statements are written which evaluate the function  $F(X_1, \dots, X_n)$  and assign the resultant value by means of an arithmetic statement to the function name. This group must include at least one RETURN statement

which returns control to the main program (ends the work of the FUNCTION subprogram). An END statement is written at the end of the FUNCTION subprogram. Note that all identifications are in force only within the given FUNCTION subprogram.

To reference the FUNCTION subprogram  $F(X_1, \dots, X_n)$ , it is enough to write the name of this function and specify their actual values (actual arguments) in place of dummy arguments. For example, to calculate the area of a rectangle by its sides, write a FUNCTION subprogram S (A, B):

```
FUNCTION S (A, B)
  S = A * B
  RETURN
END
```

Then, to calculate the area of a rectangle having sides of 3.7 and 2.6, it is enough to write S (3.7, 2.6) in the program.

Besides numerical constants, actual arguments in the FUNCTION subprogram may be any arithmetic expressions. Variables included in an arithmetic expression must be assigned numerical values at the time the FUNCTION subprogram is referenced to.

Note that both simple variables and arrays may be used in the FUNCTION subprogram either as dummy, or natural arguments. If a certain array is used as a dummy argument (parameter), it must be declared in a DIMENSION statement contained within the FUNCTION subprogram. In this case, either real numbers or simple variables may be written within the parentheses following the array name. These simple variables must be indicated in the dummy argument list of the FUNCTION subprogram in question. If a certain array is an actual argument, it must be declared in the program utilizing a given FUNCTION subprogram, the maximum permissible subscripts being specified in this declaration.

An important feature of the FORTRAN programs is that the variables and labels may be localized within separate program segments. It means that variable names and label numbers in any segment may be chosen regardless of whether similar names and labels are present in other segments.

However, in the course of a program, a necessity may arise to utilize the current value of certain simple variables and arrays from the other segments of the same FORTRAN program in certain segments. This can be done by declaring the corresponding simple variables and arrays as dummy arguments. Certain computation procedures may be written in the form of FUNCTION subprograms and subroutines, and then used in designing FORTRAN programs. Computational procedures (solution of sets of equations and the like) often used in practice and written in the form of FUNCTION subprograms

or subroutines comprise a library of standard subprograms (segments). The library of standard programs materially helps in programming new tasks to solve many complicated problems.

In conclusion, we consider a sample program in FORTRAN.

**Example.** A program of tabulating a rational fraction prints a table of the values of the following function:

$$f(x) = (a_n x^n + a_{n-1} x^{n-1} + \dots + a_0) / (b_m x^m + b_{m-1} x^{m-1} + \dots + b_0)$$

provided that the polynomial powers in the numerator and denominator of the fraction are not greater than 100.

Let the source data be punched into cards in the following sequence: AMIN, AMAX, H, N, M. The factors of the numerator occupy N cards, while the factors of the denominator M cards. AMIN stands for the minimum value of an argument, and AMAX for the maximum value of an argument. H is a table spacing (step).

The program itself takes the following form:

```

      DIMENSION A(100), B(100)
1   FORMAT (3F9.4,2I3)
2   FORMAT (F9.5)
3   FORMAT (5X, F9.4, 10X, F9.4)
4   FORMAT (5X, F9.4, 10X, 'INDEFINITE')
      READ (3,1)AMIN,AMAX, H, N, M
      READ(3,2) (A(I), I = 1, N)
      READ(3,2) (B(I), I = 1, M)
      X = AMIN
12  IF(AMAX—X) 10, 20, 20
20  F=GORN (X, M, B)
      IF(F) 11, 21, 11
11  F=GORN(X, N, A)/F
      WRITE(2,3) X, F
      X = X + H
      GO TO 12
21  WRITE(2,4)X
      X = X + H
      GO TO 12
10  STOP
      END
      FUNCTION GORN (X, N, A)
      DIMENSION A(100)
      GORN = 0
      DO 5I = 1,N
      J = N + 1 — I
5   GORN = GORN*X + A(J)
      RETURN
      END
```

The result will be printed out in two columns: the left column is for the argument value and the right column for the function value.

The program consists of two segments represented by the basic segment and a FUNCTION subprogram computing the value of the polynomial by the Horner's method. The initial instructions provide for reading the source data from the cards. The loop is organized according to the step of argument change (instruction 12 and previous). This done, the value of the function for a successive value of the argument is computed. If the denominator is equal to zero, a word INDEFINITE is printed on the table line in place of a function value.

## **CHAPTER 10**

# **INTRODUCTION TO OPERATING SYSTEM DOS/ES**

## **10.1. DOS Components and Structure**

As with other operating systems, the Disk Operating System (DOS) is used for making the process of program creation automatic, cutting down the time interval between the problem statement and the result output, and also for improving the computer efficiency.

The DOS/ES is intended for helping all those who are involved in the use of computers. They are programmers, operators and maintenance personnel.

The DOS/ES offers the programmer a wide set of tools which allow the problems involved to be divided into parts, which then are coded in different languages and combined into a program ready for execution. The operating system includes features of automatic program debugging, and also standard input/output procedures and a set of system programs for handling data.

The DOS/ES offers the computer operator great possibilities to control the computation procedure, the operator's actions being minimized.

The system supplies attending personnel with information to allow permanent monitoring of the computer condition.

The DOS/ES consists of software components which form two groups: a control program performing the functions of job control, program execution control and data management, and processing programs that form the programmer's service. Though included in the operating system, the processing programs are executed exactly in the same manner as problem-state programs, i.e. programs written by the user.

As to its functions, the control program is subdivided into the following parts:

An *initial program loader* which prepares the DOS/ES for operation;

A *supervisor* which manages the whole of processing on a computer;

A *job control program* which automates job-to-job transitions in the batch processing mode;

A *single program initiation routine* (Single Program Initiator) which performs the preparation for execution of single programs in the multiprogramming mode;

An *I/O control system* which is software designed to organize data transfer between the main storage and peripheral units.

The system processing programs include:

*Compilers of high-level languages*, such as PL/I, FORTRAN, COBOL and also of the Assembler language;

A *linkage editor* which combines separately compiled portions into an executable program;

A *librarian* which is routine designed to maintain, service and organize the various DOS/ES libraries;

A *debugging routine* which provides the programmer with tools of automatic program debugging;

*Re-writing routines* and other service facilities for copying, relocation, printing, punching, and other auxiliary operations on data file;

*Sorting routines.*

The DOS/ES is a modular structure which allows the user to adapt the system to actual ES computer hardware configuration and nature of problems being solved. Certain software components of the DOS/ES and also the functions of the control program may be included in the system at the user's option.

The process of assembling, compiling and linking the constituents of an actual DOS/ES version is called the *system generation*. During system operation the DOS/ES components are disk resident. The disk pack containing the DOS/ES is known as the *system residence*.

## 10.2. Basic Concepts of DOS

**The job.** In order to enable a computer to carry out certain work, the operating system must be given (by the operator or programmer) a job. From the standpoint of organizing computer actions, a job is the smallest unit of work that can be presented to the computing system by a user. Each job is considered as independent of other jobs and this allows parallel execution of jobs.

A job is declared with the aid of control statements each of which is punched into a separate card and defines the name of the job, beginning and end of the job, the name of the program which is to be executed.

At the programmer's option, a job can be broken down into seven

ral subdivisions called steps executed in sequence. For example, a job involving a compile operation, an editing operation, and program execution consists of three steps: a compile step, an edit step, and a go step (execution of the program). A most simple job may have one step.

The control program organizes the receipt of jobs, their tests, preparation of the requested programs for execution, their starting, and automatic branch to the next job. All these functions go under the general term *job control*.

**The task.** Each time the control program has recognized a job step, it is accepted as a task, i.e. a piece of work to be carried out by the program specified in the step of the job. To this end, the system must allocate the required resources: main storage and external storage, input/output devices, processor time, etc.

The task is under control of the Supervisor. A task may be executed in the single program or multiprogramming mode. In the single-program mode the computer solves only one task, and all resources of the computer are at the disposal of this task. In the multiprogramming mode, several independent tasks may be handled concurrently, and the operating system resources must be shared by these tasks. The DOS/ES provides concurrent execution of up to three programs.

**The logical units.** The ES EVM computers permit attachment of many diverse input/output units. The peripheral units used vary with the kind of the computer. Physical addresses of the units may also vary either on computers furnished with a similar set of units, or, the more so, on computers of different configurations.

To make the DOS/ES software components and user's programs independent of the actual physical addresses of the units, use is made of the logical unit block, or LUB table containing a standard set of symbolic names which the programmer uses each time he refers to a peripheral device. Therefore, specified in the program is a logical unit, rather than an actual physical address. Directly before execution of the program on a computer, the programmer or operator assigns the logical unit a physical address, thus defining the correspondence between the logical and physical units.

The symbolic names of the logical units take the form `SYSxxx`, where `xxx` may take either certain alphabetic symbol or a numerical value from 000 to the maximum number of logical units serviced by the system.

Certain logical units are used for the work of the DOS/ES itself, for which reason they are known as the system logical units. The names of these units are tabulated in a fixed sequence:

<code>SYSRES</code>	— system residence area;
<code>SYSRDR</code>	— input unit for job control statements;
<code>SYSIPT</code>	— input unit for application programs (system input);

- SYSPCH     — unit for punched-card output (system punch);
- SYSLST     — unit for printed output (system printout);
- SYSLOG     — unit for operator messages

The system logical units used are essential to the work of the control program. There are other system logical units which are employed by various DOS/ES components. The logical units SYS000—SYS $n$ nn are known as the logical units of the programmer.

**The library.** In the Operating System DOS/ES a program written in any programming language is called the source. It contains the source texts of the programs which are translated by the corresponding compiler. The result of the translation operation is an object module whose format is already independent of the source programming language, i.e. it is common to all compilers of the system. However, the object module is not yet the machine program. It cannot be used directly for execution and must undergo one more stage, namely: editing. In this stage, the object modules are processed by the linkage editor as it builds program phases or load modules suitable for loading into storage and execution.

Depending upon the stage they are in, all the DOS/ES programs may be stored in a library of one of the following three types: a source-statement library, a relocatable library, or a load module library.

Presence of the relocatable library is a pre-requisite for functioning of the DOS/ES, while the other two libraries are required if a given machine performs operations on creation and debugging of new programs. However, common practice is to have all the three libraries. The DOS/ES system libraries are represented by a single file called the resident volume and contained in the system residence.

In addition to the system libraries, the DOS/ES may include private libraries which are arranged on disk packs and may be contained either in the system residence, or on other packs. It is good practice to use private libraries in the debugging stage, and also for debugged programs of narrow use, for the size of system libraries is limited and the programmer should not overcrowd them with private programs of no interest for other users of the computer.

### **10.3. Job Control Program**

The Job Control Program accomplishes the receipt of input job stream and prepares the system for execution of the job received. The program phases are permanently resident in the relocatable library in the system residence, and are called into main storage by the supervisor on completion of the previous job or job step, and also after the system initial load procedure. Execution of the Job Control Program generally terminates with control return to the supervisor

which loads the required program from the load module library in the main storage.

To perform its functions the Job Control Program receives information from the job control statements prepared by the programmer. These functions, in particular, include assigning the logic units physical units, describing the requirements to the operating system, handling information on the labels of volumes and files, preparation of the program for execution.

Given below are the formats of the basic job control statements, only the main, most often used parameters being specified in them. Specific means used but scarcely are omitted.

**Begin the job.** The JOB statement must be the first in any job. It specifies the beginning of job control strings of a new job. The format of this statement is as follows

//JOB *job-name*

The two slashes must appear in columns 1 and 2 of the control statements. One or more blanks separate the slashes from the keyword 'JOB' which identifies the operation code (JOB in this event), and one or more blanks separate operands. The only operand of the JOB statement is *job-name* which may consist of from one to eight alphanumeric characters. Comments (accounting information) are optional and may be written after the *job-name* being separated from it by one or more blanks.

**End the job.** This statement is the last one in a statement packet and has a format somewhat different from the other control statements:

/&

The slash and ampersand occupy columns 1 and 2. Column 3 must contain a blank. Comments may be written starting with column 4.

**Assign a physical device.** The ASSGN statement is used to assign a logic unit an actual physical device. The assignment carried out by the Job Control Program on an ASSGN statement holds for this job only until a new assignment for the same logic unit is made, or until the end of a current job. The format of the statement is as follows

//ASSGN SYS*xxx*,X'*cuu*'

The operands of the ASSGN statement are the symbolic name of the logic unit and the location of the physical device assigned to it. The latter is written in the hexadecimal code: *c* stands for the channel number and *uu* for the number of the unit in the channel.

**Specify options.** The OPTION statement specifies the modes of executing the Job Control Program and certain other control prog-

rams. The specified options hold until a new **OPTION** statement is encountered to specify an opposite mode, or until the current job is completed. Standard options defined by the computer operator or during the system generation are set at the beginning of any job. The statement format is as follows

// **OPTION** *option* = 1, [, *option* = 2, . . .]

The basic options which may be specified in the field of operands are:

**LINK** indicates that the object module must be processed by the Linkage-Editor. In this option all compilers write the object modules formed in a special area of the disk unit;

**NOLINK** cancels the **LINK** option. The compilers can also cancel if the source module contains errors hindering successful execution of the problem-state program;

**CATAL** causes the program phases formed by the Linkage-Editor to be cataloged in the load module library. Specifying the **CATAL** option causes a **LINK** option to be set;

**DECK** causes **SYSPCH** to punch the object modules formed by the compiler;

**NODECK** cancels the **DECK** option;

**LIST** specifying this option makes the compilers produce the source module listing on **SYSLIST**;

**NOLIST** cancels the **LIST** option.

The options selected may be written in any order.

**Execute the program.** The **EXEC** statement must be last in a job step. This statement identifies the particular problem-state program whose execution is to be started. The general form of this statement is:

// **EXEC** [*program-name*]

In the DOS/ES operating system any program is called for execution from the load modules. There are two types of program storage in a library. Programs cataloged in a library are permanently resident in it, and are called for execution by their names. In specifying the **LINK** option the phase formed by the Linkage-Editor is temporarily loaded in the load module library and stored therein only during the execution of a current job. Such a program must be executed at once after editing and is called with the aid of an **EXEC** statement with no program name.

#### 10.4. Linkage-Editor

This program produces executable programs of object modules created by the DOS/ES compilers. The editing is an essential completing stage of the process during which programs are prepared for execution.

The DOS/ES implemented stagewise preparation of programs, including the editing process, represented a flexible universal apparatus. The programmer is in a position to break up a program into parts, choose a programming language most suitable for each of these parts, translate each part regardless of each other, and then edit the obtained object modules into an executable program.

All links existing among various independent portions of a program are defined by the programmer symbolically in source languages, and are known as external references. These are enabled, i.e. assigned an actual value only during the editing stage. Combining object modules into a program and enabling external references involved in it are main functions of the linkage editor.

Object modules may be combined by the linkage editor into a program of simple structure, or an overlay program. In case of simple structure the program consists of one phase and is called into main storage for execution as a whole. The overlay structure presupposes several phases in the program which are called for execution in turns, and succeeding phases may be called to the same area as certain previously called phases, i.e. to overlay them. The overlay structure may be either with a root phase or without it. In the first case there is a phase which is called a root phase which remains in main storage throughout the execution of the program. In case of the structure without root phase, it is supposed that each phase completed calls the next phases to its place.

The program structure is chosen by the programmer and planned in the source language. The simple structure of a program somewhat reduces the time of execution, while the overlay structure allows the size of main storage to be decreased.

The program structure chosen must be taken into account both in the program building stage and in the editing stage. During the program building stage the programmer must provide links between separate parts and methods of information transmission from part to part, using the means provided for the purpose in the source languages. During the editing stage the programmer tells the Linkage-Editor what program structure is to be used with the aid of control statements.

The ACTION statement is used to specify editing options, if the programmer needs options other than standard. All ACTION statements must be first in the input information of the Linkage-Editor.

The PHASE statement gives a name to the phase being built and specifies the address of its loading. This statement must precede all object modules which are to be included in a given phase. If no PHASE statement is used, or its specification is invalid, the Linkage-Editor builds a dummy PHASE statement and the editing is continued. However, a program with a dummy PHASE statement cannot be cataloged in the load module library. Though a CATAL opti-

on has been specified, such a program is placed in the library only for temporary storage and can be executed solely in the current job.

To declare the phase structure of the program under editing, use is made of INCLUDE statements. All INCLUDE statements read after a PHASE statement refer to the phase declared in this PHASE statement until a new PHASE statement has been read. The operand of the INCLUDE statement specifies the name of the module from the private or system relocatable library which is to be included in the phase. The operands of an INCLUDE statement may be dropped. If that is the case, the text of the associated object module must immediately follow the INCLUDE statement.

By means of an ENTRY statement the programmer may provide for an optional phase entry point, if it is other than the beginning of the first module of the first phase of the program.

In addition to the explicit inclusion of modules in a phase through the use of INCLUDE statements, the Linkage-Editor has an apparatus of automatically including object modules from a private or a system library. The automatic inclusion holds when unresolved external references are left after all the modules requested by the programmer have been combined into a phase. To resolve those references the Linkage-Editor automatically calls the required object modules from the library. The automatic inclusion may be cancelled on a request from the programmer.

---

## REFERENCES

1. Calderbank, V. G. *Course on Programming in FORTRAN IV*. Chapman and Hall, London, 1972.
2. Germain, C. B. *Programming the IBM/360*. Prentice-Hall, Inc., Englewood Cliffs; New Jersey, 1967.
3. Gruend F. *FORTRAN-IV-Programmierung*, Berlin, 1972.
4. *COBOL, ES EVM*. Moscow, 'Statistika', 1978 (in Russian).
5. *FORTRAN, ES EVM*. Moscow, 'Statistika', 1978 (in Russian).
6. Lebedev, V. N. *Introduction to Programming Systems*. Moscow, 'Statistika', 1975 (in Russian).
7. *Programming in ES EVM Assembler*. Moscow, 'Statistika', 1975 (in Russian).
8. Scott, R., Sondak, N. *PL/I for Programmers*. Addison-Wesley Publishing Company, Menlo Park, California, London, Don Mills, Ontario.

# INDEX

Absolute module 194  
ALGOL 239  
Algorithm 45-48  
Algorithmic language 51  
AND group instruction 118  
Assembler 51, 194  
Assembler translator 222  
Assumed decimal point 134

Base 10, 11, 54  
Base register 54  
Binary-coded decimal system 13, 25  
Binary number system 12  
Bit 25  
Block 38, 39, 40, 228  
Blocked records 228  
Burst mode 173  
Business-oriented computer 147  
Byte 13  
Byte mode 173

Card reader 23  
Cause code 164  
Central (internal) store 32  
Chain data flag 173  
Chain command flag 176  
Channel 23, 45  
Channel address word 179  
Channel command register 175  
Channel program 173  
Channel states word 179  
Character-string-picture 251  
Character-string constant 254  
COBOL 239  
Common field 196  
Conditional branch 67  
Condition code 64, 116, 164  
Control unit 24

Data regeneration 33  
Data shift 114  
Data storage time 35  
Decimal overflow 127  
Decision block 5, 63  
Declarative (description) I/O macro 228  
Default attribute 249  
Default value 249  
Destructive storage 33

Digit 9  
Digit selector (digit select character) 148  
Directory 217  
Disk operating system 319  
Double precision number 57, 103, 108

Editing pattern 148  
Effective address 54, 90  
Exclusive OR group instruction 121  
En ry-poin -name 226  
Explicit address constant 216  
Explicit addressing 203  
External interrupt 105  
External name 216  
External store 32  
External symbol directory 221

Field separation character 149  
File description table 228  
File name 229  
Fill character 148  
Fixed-point binary constant 248  
Fixed-point decimal constant 247  
Fixed-point instruction 97  
Fixed-point (natural) representation 15  
Floating-point binary constant 249  
Floating-point decimal constant 248  
Floating-point instruction 102, 111  
Floating-point register 102, 109  
Floating-point (semilogarithmic) representation 16  
Flowchart 48, 69, 95  
FORTRAN 51, 238, 301-319

Hexadecimal constant 212  
Hexadecimal number system 12

Identification field 195  
Identifier 51  
Implicit addressing 204  
Index 54  
Index register 54  
Initial program loading 160  
Input/output block 49  
Input/output channel 172  
Input/output controller 24

- Input/output interface 40, 42, 43, 45
- Input/output processor 23
- Instruction length code 164
- Interface 21
- Interpreter 194
- Interruption code 164
- Iteration loop 85
- Label 51
- Length attribute 197
- Library 322
- Linkage micro instruction 227
- Linkage register 79
- Literal 199
- Local storage 32
- Long-term storage 32
- Loop 84
- Loop with address modification 88
- Loop with counter 86
- Machine check interrupt 165
- Machine-operated programming language 237
- Macro 194, 225
- Macro definition 225
- Macro expansion 226
- Macro library 225
- Magnetic core storage 33
- Magnetic disk drive (unit) 35, 52
- Magnetic tape drive (unit) 36-39
- Main program 80, 81
- Main storage 24, 25; 32, 33, 53
- Master control program 160
- Memory 23
- Multiplex mode 173
- Multiplexor channel store 32
- Multiprogramming 173, 174
- Multiway branching 68
- Name field 195
- Nondestructive storage 33
- Nonpositional number system 9
- Normalized number 16
- Object module 194
- Ones complement representation 19
- Operand 53, 229
- Operands field 195
- Operation code 53
- Operation field 195
- OR group instruction 119
- Overflow 64
- Packed decimal format 29, 30
- Packed decimal operand 126, 129, 130-131, 136
- Parties 45
- Peripheral device 23, 43, 44
- PL/I 51, 240-300
- Positional ('place value') number system 10, 11
- Predefined block 50
- Privileged instruction 162
- Problem-oriented programming language 238
- Process block 49
- Processing state 163
- Processor 21
- Program-controlled interruption flag 176
- Program interrupt 165, 166
- Program linkage means 224
- Program listing 194
- Program mask 164
- Protection key block 32, 165
- Program status word 162
- Protection mode 165
- Selector channel 173
- Self-defining term 198, 199
- Sense byte 177
- Sequential access storage 34
- Significance indicator 149
- Significance stater 148
- Significant exception 107
- Single-precision number 103
- Skip flag 176
- Source module 194
- Statement continuation column 195
- Statement field 195
- Status byte 184
- Storage device access time 235
- Storage device capacity 34, 35
- Storage key 165
- Storage matrix 33
- Storage protection key 163
- Subprogram parameter 110
- Subroutine 73, 74, 80-82
- Supervisor 160
- Supervisor call interrupt 11
- Suppress-length indicator 176
- System mask 163
- Timer interrupt 165
- Translation table 155
- Translator 194
- Twos complement representation 19
- Unblocked record 228
- Unconditional branch 64, 67
- Unnamed control section 222
- Zoned (unpacked) decimal format 30, 31, 141





